TURING 图灵原创

《保卫萝卜2》
素材官方授权使用

Cocos2d-JS 游戏开发

凌建风 著



林志斌 飞鱼科技联合创始人
王哲 Cocos引擎创始人
林顺 Cocos引擎联合创始人
联合作序

凌华彬 panda Cocos2d-JS引擎核心开发者
屈光辉 子龙山人 Cocos2d-x引擎核心开发者
卞安 业界专家 红孩儿
李华明 业界专家 Himi
CocoaChina社区和CVP平台
林德辉 GameRes游资网CEO
倾情推荐

数字版权声明



凌建风

飞鱼科技国际有限公司客户端程序员，触控科技认证CVP，Cocos2d-JS方向的权威导师。从毕业到现在，一直从事Cocos2d-x和Cocos2d-JS游戏开发，先后在Cocos引擎中文官网和全球最大苹果开发者社区CocoaChina发表多篇博文以及两套Cocos2d-JS视频教程。

TURING 图灵原创

Cocos2d-JS游戏开发

凌建风　著

人民邮电出版社
北　　京

图书在版编目（ＣＩＰ）数据

Cocos2d-JS游戏开发 / 凌建风著. -- 北京 : 人民邮电出版社, 2016.5（2016.11重印）
（图灵原创）
ISBN 978-7-115-42148-7

Ⅰ. ①C… Ⅱ. ①凌… Ⅲ. ①移动电话机－游戏程序－程序设计②便携式计算机－游戏程序－程序设计 Ⅳ. ①TN929.53②TP368.32

中国版本图书馆CIP数据核字(2016)第083126号

内 容 提 要

本书结合《保卫萝卜 2》以及多个实例详细介绍了 Cocos2d-JS 游戏引擎，书中共分为 4 个部分，第一部分为基础篇，主要介绍了 Cocos 引擎家族史、各平台下的环境搭建、开发工具的选用，还有引擎的核心框架、动作模块、事件机制、音频处理以及屏幕适配等；第二部分为进阶篇，内容包括数据存储、粒子系统、UI 控件、性能优化以及游戏地图；第三部分为高级篇，主要涉及与其他语言的反射调用、Chipmunk 物理引擎、网络编程以及 JavaScript Binding；第四部分为实战篇，即《保卫萝卜 2》实战。

本书适合有一定 JavaScript 语法基础，并且想快速系统学习 Cocos2d-JS 游戏开发的人员阅读。

◆ 著　　　　凌建风
　责任编辑　王军花
　责任印制　彭志环
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　印刷
◆ 开本：800×1000　1/16
　印张：21.75
　字数：514千字　　　　2016年 5 月第 1 版
　印数：3 001 – 3 400册　　　2016年 11 月北京第 2 次印刷

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

但愿自己成长的速度

能够赶上父母老去的步伐

谨以此书，献给我伟大的爸爸妈妈

序　一

飞鱼科技发布的多款游戏使用的都是 Cocos 引擎团队的产品，旗下产品包括《神仙道》手游版、《保卫萝卜》系列、《三国之刃》《囧西游》《小鱼飞飞》等。之所以选择 Cocos 引擎，是因为它具有优秀的跨平台能力，具有易于使用及高效、灵活的特性，让我们的研发团队可以把更多精力放在游戏设计本身上，大大提高了开发效率，降低了研发成本。

作为忙碌的游戏开发者中的一员，建风在工作之余完成了这本书。书中结合项目开发中的实践经验，以读者的视角精心准备，通俗易懂，力求最佳的阅读和学习体验，让更多的开发者从实战角度出发，更加全面地了解 Cocos2d-JS 引擎的特性。在建风提到撰写这本书时，飞鱼科技在第一时间给予了支持，将《保卫萝卜》的相关美术素材授权给他在书中使用。我们希望能通过大家熟悉的产品，使读者更加直观和生动地学习 Cocos2d-JS 引擎。

创作本书历时近一年，内容和编排诚意十足，由浅入深的实例教程降低了学习门槛，值得分享给想要学习和了解 Cocos2d-JS 引擎的开发者。

序　　二

几年来，在 Cocos 引擎团队的坚持和专注下，Cocos 引擎经历了一次次突破，越来越得到市场的认可和无数开发者的支持。在国内 iOS、Android、微信游戏等分发渠道上，近半数的游戏产品均选择使用 Cocos 引擎开发，同时也涌现了大量如《刀塔传奇》《梦幻西游》《怪物弹珠》等月流水过亿的产品，这是让我们很自豪的一件事。

Cocos2d-JS 是 Cocos 引擎的分支版本，同样有非常亮眼的成绩，比如口碑收入双丰收的《三国杀传奇》《航海王启航》，甚至是近年越来越受到关注的独立游戏领域，也诞生了诸如《锻冶屋英雄谭》《超级幻影猫》等众多得到了苹果青睐的产品。

易用、低门槛、跨全平台、热更新支持……Cocos2d-JS 的诸多优点毋庸置疑，加之 Cocos 团队新推出的创新性开发工具 Cocos Creator，脚本化已经是一种趋势，Cocos2d-JS 将在这方面为开发者提供更有力的帮助和支持。

建风是我见过的对 Cocos 引擎有着非常深入了解的 90 后开发者之一。在本书中，从最基本的模块介绍到每个方法的详细调用，他融合了自己对 Cocos2d-JS 的理解，对这款引擎进行了非常全面的剖析和阐释，深入浅出，有着巨大的参考价值。他还获得了飞鱼科技《保卫萝卜》系列素材的授权，并在书中有所展示。本书结合《保卫萝卜 2》的实战项目，对 Cocos2d-JS 的具体使用进行升华，推荐一读。

原计划半年写完本书，但建风追求精益求精、精雕细磨的态度让这个时间延长到了 10 个月。希望这本书能帮助更多的开发者更好地理解 Cocos2d-JS 这款功能强大的引擎，也希望它能在带来大量干货的同时帮助更多开发者产出更优质的产品。



序　三

早在2012年，Cocos引擎团队在Google的赞助下成功将Cocos2d-x移植到了Cocos2d-HTML5版本，成为全球最早的HTML5游戏引擎之一。2014年年初，引擎团队推出了Cocos2d-JS，实现了Cocos2d-HTML5和Cocos2d-x JSB引擎框架的彻底融合，统一了手机页游和原生游戏的开发工作流，提供了优秀的跨平台部署能力。

目前，引擎团队基于Cocos2d-JS推出了创新性开发工具Cocos Creator。该编辑器具备彻底脚本化、组件化和数据驱动的特点，并以内容生产为核心，适合团队不同角色进行高效的合作以及开发。

从引擎功能增强、性能优化，再到工具链的强化和升级，Cocos2d-JS一直以来都致力于为行业和开发者铺平道路。

本书对Cocos2d-JS引擎进行了很全面的剖析和讲解，从模块、框架的拆解到各类方法的详细调用和开发中的各种优化技巧，内容由浅入深，具有很好的指导和参考作用。同时，建风获得了飞鱼科技《保卫萝卜》系列素材的授权，并在书中对这款很早便非常流行的休闲领域标杆产品进行了全方位的解析。同时，书中也通过介绍《保卫萝卜2》各个模块的开发来进行Cocos2d-JS实战引导，具有非常大的实际参考价值，值得一阅。

正如建风所说，本书其实更像一款“十月怀胎”诞生的产品，书中所有内容都进行了精雕细磨，我相信它能为更多的开发者带来巨大的实用价值，让Cocos2d-JS成为开发者手中的开发利器。



前　　言

近几年，手游行业的发展可谓是百家齐放、万紫千红。短短两年，便出现了多款月流水过亿，甚至是近十亿的产品，例如《刀塔传奇》《梦幻西游》《怪物弹珠》等，而这些游戏都是采用 Cocos 引擎开发的。

Cocos2d-JS 作为 Cocos 引擎的下一个分支，从 2014 年 3 月份发布 3.0 Beta 版到现在已经经历两年的磨练了。这两年来，Cocos2d-JS 取得的成绩不容置疑。再加上 Cocos Creator 编辑器的推出，Cocos 引擎面向开发者脚本化已经成为趋势，而 Cocos2d-JS 正是官方现在主推的方案。

本书编写始于 2015 年 4 月，原计划 6 个月撰写完成，但直到 2016 年 1 月才最终完成，历经 10 个月。我常常调侃自己，这本书是我“十月怀胎”生出来的。在这 10 个月中，我过着机器般的生活，除了上班就是写书，偶尔有朋友约我周末一起出去玩，我只能问他们一句：“周末是什么，能吃吗？哪里有卖？多少钱一斤？”这样装疯卖傻，只是为了写出一本对得起自己、对得起读者的书。

在保证图书内容的同时，我更注重它阅读起来是否轻松愉快、通俗易懂、能够现学现用，所以，我对书中所有的内容都进行了精雕细磨，哪怕它只是一张图片、一行注释或者一小段代码，都精益求精。对我来说，这本书更像是一个产品，我尽可能地站在用户的角度去撰写和设计这本书，由衷地希望这本书能给广大开发者带来帮助，如此，我便心满意足了。

本书结构

本书共 4 部分，17 章。

第一部分为基础篇，共 7 章，各章简述如下。

第 1 章“Cocos2d-JS 介绍”：这一章介绍了 Cocos 引擎的家族史以及引擎的相关创始人，说明了 Cocos2d-x 和 Cocos2d-JS 之间的关系，详细介绍了 Cocos2d-x 引擎包的目录结构。

第 2 章“Hello World”：这一章介绍了 Cocos2d-JS 项目的目录结构，演示了环境搭建，讲述了如何通过 Cocos Console 创建、编译、运行和打包 Cocos2d-JS 工程。此外，还介绍了如何使用 Cocos DevTools 工具提高学习效率。在这一章中，你还可以了解到 Cocos2d-JS 游戏的启动流程。

第 3 章“核心框架”：这一章介绍了 Cocos2d-JS 核心框架相关的类，如 `cc.Director`、`cc.Node`、`cc.Scene`、`cc.Layer`、`cc.Sprite`、`cc.Label` 等，还讲解了 BMFont 编辑器的用法。最后，我们开发了《保卫萝卜 2》的主页面。

第 4 章“动作模块”：这一章介绍了 cc.Action 动作父类以及瞬时动作和持续动作，其中持续动作又包含属性变化动作、视觉效果动作以及复合动作。除此之外，还介绍了变速动作。最后，我们让《保卫萝卜 2》主页面动了起来。

第 5 章“事件机制”：这一章介绍了事件管理器、触摸事件、加速计感应事件、键盘事件、鼠标事件、自定义事件等。最后，在这一章的实例中，封装了一个虚拟摇杆。

第 6 章“音频处理”：这一章介绍了 cc.audioEngine 的常用 API，列出了 Cocos2d-JS 在各平台下支持的音频格式，并讨论了背景音乐应当在何时播放最为合适。最后，给《保卫萝卜 2》添加了相关的音频。

第 7 章“屏幕适配”：这一章介绍了屏幕适配原理以及 5 种系统预设适配模式，分别是 SHOW_ALL、NO_BORDER、EXACT_FIT、FIXED_HEIGHT、FIXED_WIDTH，并交代了官方目前推荐的适配方案。

第二部分为进阶篇，共 5 章，各章简述如下。

第 8 章“数据存储”：这一章介绍了 cc.sys.localStorage 的用法，以及 JSON 和 plist 文件的读取。最后，完成了解锁《保卫萝卜 2》“天天向上”玩法的开发。

第 9 章“粒子系统”：这一章介绍了 cc.ParticleSystem 类，并将 cc.ParticleSystem 类的属性划分为粒子配置、发射器类型、重力配置、径向配置、发射器位置、粒子纹理、粒子颜色以及混合函数等，还介绍了 ParticleDesigner 和 ParticleEditor 粒子编辑器。

第 10 章“UI 控件”：这一章介绍了 Cocos2d-JS 中的 UI 控件，例如文本、编辑框、按钮、复选框、滑块、加载条、布局、滚动视图、列表视图以及分页视图等。

第 11 章“性能优化”：这一章介绍了 Cocos2d-JS 游戏开发的一些优化技巧，例如对象缓冲池 cc.pool 的使用，OpenGL/WebGL 下可以使用批处理 SpriteBatchNode 进行优化，在 Canvas 中可以将不活跃的 Layer 烘培起来。在资源管理方面，介绍了 SpriteSheet，SpriteSheet 可以通过 TexturePacker 工具制作，并在这一章的最后抛出了 TinyPng 在线图片压缩云应用。

第 12 章“游戏地图”：这一章介绍了 Tiled Map 编辑器的用法，还讲解了 cc.TMXTiledMap、cc.TMXLayer 以及 cc.TMXObjectGroup 类。

第三部分为高级篇，共 4 章，各章简述如下。

第 13 章“反射调用”：这一章介绍了如何在 JavaScript 中通过反射机制调用 Objective-C 和 Java 代码。

第 14 章“Chipmunk 物理引擎”：这一章介绍了 Chipmunk 物理引擎中的基本物理概念，以及刚体、形状、约束、空间、碰撞检测、查询等。最后，做了一个拖动刚体的效果。

第 15 章“网络编程”：这一章介绍了 OSI 参考模型和 TCP/IP 参考模型，以及基于 HTTP、WebSocket、SocketIO 的通信。最后，完成了一个基于 Node.js + Scoket.IO 的在线聊天系统。

第 16 章“JavaScript Binding”：这一章介绍了 Cocos 引擎中的脚本绑定框架以及自动绑定，并深入探究绑定技术，一步一步演示如何绑定自己的 C++类。最后，实现了 SQLite3 的绑定。

第四部分为实战篇，共 1 章，简述如下。

第 17 章“塔防游戏——保卫萝卜 2”：这一章介绍了《保卫萝卜 2》各个模块的开发，例如

关卡设定、出怪逻辑、炮塔生成、路线生成、碰撞检测、游戏对象管理等。

读者对象

本书适合有一定 JavaScript 语法基础的读者。因为本书并没有讲解 JavaScript 语法，如果读者不熟悉 JavaScript 语法，那么我建议你先去阅读《JavaScript 高级程序设计》或者《JavaScript 面向对象编程指南》等书。

除此之外，再无其他要求，你可以是完全不懂 Cocos 引擎的初学者，也可以是有 Cocos 引擎其他分支（例如 Cocos2d-x 和 Cocos2d-Lua）开发经验的开发者，甚至是有一定 Cocos2d-JS 游戏开发经验的开发者。

关于源代码

书中所有的源代码以及《保卫萝卜 2》的实战代码均可在图灵社区本书主页（http://www.ituring.com.cn/book/1783）或我的 Github 仓库 https://github.com/lingjianfeng 中下载。

本书说明

- 为了方便读者，本书每一章的参考资源、相关工具的下载地址等都放在随书源码中，你可以直接打开源码，将参考资源或下载地址复制到浏览器中直接打开，而不用自己手动输入。
- 为了使读者能更快、更好地阅读代码，本书中的所有代码保持了代码高亮样式，并且采用表格的方式，明确标注出行号。其中，JavaScript 代码高亮样式与 WebStorm 保持一致，C++代码与 Xcode 保持一致，Java 代码和 Eclipse 保持一致。虽然本书不是彩色印刷，但是在黑白印刷上，代码依旧可以通过粗细来达到“高亮”效果，但愿我这样的坚持能给读者带来更好的体验。
- 为了突出重点，不必要显示的代码我用//……替代，表示代码省略。
- 为了节省版面空间，书中部分图片经过 Photoshop 处理，大部分裁剪掉了一些没用的空白区域等，这并不会影响读者理解相关知识点。
- 为了提高代码注释的可阅读性，我在代码注释中保留了我个人的注释风格，采用中括号括起关键词，起到“高亮”效果，例如：// 加载[主页面]。
- 本书所有软件默认采用 Mac 版。
- 本书基于 Cocos2d-JS 3.9 版本编写。
- 随书项目《保卫萝卜 2》中的所有素材为飞鱼国际科技有限公司授权于本书作者，读者可以使用这些素材来学习 Cocos2d-JS 引擎，但是不可将其用在商业途径上，若有违反，飞鱼国际科技有限公司将追究相关人员的法律责任。

勘误信息

我们尽最大的努力确保正文和代码没有错误，但是人无完人，金无足赤，错误在所难免。如果读者发现书中的任何错误，例如错别字或者是代码片段无法运行等问题，我们都希望你能及时给我们反馈，你提交的勘误不仅能够让其他读者受益，也能帮助我们进一步提高图书的质量。

读者可以在图灵社区本书主页（http://www.ituring.com.cn/book/1783）的“勘误”页面提交勘误，编辑将会第一时间处理。此外，你还可以给我发送邮件，我的邮箱是 183720950@qq.com，邮件主题请注明“图书勘误”，我将在收到邮件后的第一时间回复。或者，也可以在新浪微博中联系我，我的微博：@凌建风。

另外，也可以加 Cocos2d-JS 技术交流群（群号：186828918），与其他读者一起交流。

致谢

2014 年 6 月 22 日，我从福建师范大学信息技术学院毕业后，便选择了 Cocos 游戏开发。作为游戏行业的新人，我很感谢在这一年多来一直帮助我、陪伴我的同事们，从他们身上我学到了很多知识，也成长了许多。另外，特别感谢我现在任职的飞鱼科技国际有限公司将《保卫萝卜》系列素材授权于我，也感谢触控科技将封面素材授权于我。

感谢 Cocos2d-x 引擎核心开发者屈光辉（子龙山人）对本书的写作风格给予的指导，并提供技术支持和部分章节技术校稿。

感谢 Cocos2d-x 引擎物理引擎集成作者余波提供技术支持并对部分章节进行技术校稿。另外，本书第 14 章大部分为余波编写，而我更多的则是负责语言组织、代码实现等工作。

感谢我的好友曾彬思对本书的写作提供了帮助以及校稿等工作。

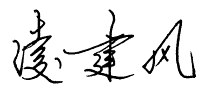
感谢我的好友邱雷对本书的写作提供了帮助以及给予我精神上的鼓励。

感谢图灵的编辑王军花对本书进行文字校稿，并给予了我很多建议和写作技巧，而且在我一次一次拖稿的情况下，不但没有催稿，还一直鼓励我。

感谢我的室友们对我的理解和包容，因为我常常写到凌晨一两点，又或者是早上 6 点多爬起来写书，这无疑对他们的作息造成了一定的影响。

最后，感谢其他朋友们对我的支持和帮助，这些人中，有的给书提供技术支持，有的给予我精神上的鼓励，让我完成了本书，他们是（按照拼音排序）：

陈龙（cnsoft）、陈志敏、傅思杰（偶尔 e 网事）、郭玉海、黄靖权、黄毅霖、江龙（龙辰忆梦）、林文海、邱磊（NetFly）、任珊（renshan）、唐梓文、徐斌杰、叶湘楠、尤泽钦、郑为满、钟东、朱圣樑、卓文理。



2016 年 1 月于厦门

目　录

第三部分　高级篇

第四部分　实战篇

第一部分

基　础　篇

第 1 章 Cocos2d-JS介绍

当今世界，互联网风起云涌。移动设备的普及，更是让手游行业发展迅速，从《神庙逃亡》到*Flappy Bird*，再到《纪念碑谷》等，这些休闲游戏曾一时间风靡全球，可谓是赚得金箔满盆。而由《刀塔传奇》《全民奇迹MU》以及《梦幻西游手游》等组成的重度游戏明星队伍更是无比的辉煌。2014年，《刀塔传奇》创下了月流水2.8亿的传奇。2015年，《全民奇迹MU》也以首月2.42亿的流水造就了奇迹；同年，《梦幻西游手游》更是以近9亿的月流水打造出了其自身在手游行业的梦幻王国。可见，手游行业正是如火如荼。

抛开手游的市场情况不说，对于游戏开发者而言，选择一款优秀的游戏引擎是开发优质游戏最根本的前提。值得一提的是，前面提到的《刀塔传奇》和《梦幻西游手游》这两款2D明星产品皆是采用Cocos游戏引擎打造而成的。

近几年，各大游戏引擎可谓是白家争鸣，有的在历史长河中销声匿迹，有的历经万般考验仍屹立不倒。现如今，Cocos和Unity 3D分别在2D和3D引擎中脱颖而出，各领风骚，独具一格。而本书的主角Cocos2d-JS正是目前Cocos引擎全力推广的一个重要分支。

本章内容：

- ❑ Cocos引擎家族
- ❑ Cocos2d-JS介绍
- ❑ 引擎目录结构

1.1 Cocos2d 引擎家族

正所谓“吃水不忘挖井人”。记得去年我大学毕业后在宝宝巴士（福建）网络科技有限公司工作时，公司技术总监指派给我一个任务，让我研究一下LiquidFun流体引擎，并用LiquidFun做出公司一款产品的Demo。在此过程中，我遇到一个比较棘手的问题，短时间内无法解决，无奈之下找到官方人员帮忙解答。而Cocos2d-x和LiquidFun整合是由Ricardo Quesada（Cocos2d之父，后面将介绍）完成的，所以官方人员便给了我Ricardo的联系方式，让我求助于他。可当初天真无邪的我问了一句，Ricardo是谁？这看上去似乎是一件很小的事情，但是却一直藏在我心里深处。

虽然我在使用Cocos引擎，却不知道Ricardo是谁。因此，我认为在学习Cocos2d-JS引擎之前，非常有必要介绍一下Cocos引擎的先祖以及Cocos引擎的家族史。

1.1.1 Cocos2d 的诞生

2008年2月，在阿根廷Córdoba市附近一个名为Los Cocos的地方，Ricardo Quesada和他的几个朋友使用Python语言开发出了一款2D游戏引擎，并且根据这个游戏引擎的诞生地，将其取名为Los Cocos。一个月之后，他们便发布了引擎的release 0.1版本，并正式将该引擎更名为Cocos2d。

不久之后，苹果公司正式成立App Store，并且发布了SDK，大量的开发人员被App Store所吸引，各种各样的应用和游戏上了iOS平台。Ricardo Quesada团队把握住了时机，在2008年6月宣布将接入iPhone平台，并于当月就发布了以Objective-C为基础的Cocos2D for iPhone 0.1版本，此版本延续了Python版的框架和设计思路。随着iOS用户越来越多，iPhone游戏也成为了用户最爱的应用。

到2010年9月，Cocos2D引擎已经在游戏开发者中流行。许多开发者第一次接触Cocos2d for iPhone引擎，此版本是Cocos2D引擎当中的明星产品。由于Cocos2d for iPhone非常成熟且功能很完善，它在开发者中得到广泛传播并使用。随后*StickWars*成为第一款在美国区付费榜夺得冠军的Cocos2D游戏，这宣告了Cocos2D引擎时代的到来。同时，英国的设计大师Michael Heald为Cocos2D设计了新的logo，如图1-1所示（此前Cocos2D的logo是一个奔跑的椰子），图1-2和图1-3分别为Cocos2d-x早期logo和Cocos2d-x引擎现在的logo。



图1-1 Cocos2D logo



图1-2 Cocos2d-x早期logo



图1-3 Cocos2d-x现在logo

也正是在那个时候，Cocos2D社区的开发者开发出了最早的周边工具——Zwoptex和Particle Designer。关于Particle Designer的更多信息，可参见第9章。

> **说明** Zwoptex为早期的精灵表制作工具，现在更常用的精灵表制作工具为TexturePacker，详情可参见第11章。

1.1.2　Cocos 引擎家族大事记

2010年11月，来自中国厦门的开发者王哲以及他的团队基于Cocos2D制作出了Cocos2d-x引擎。Cocos2d-x引擎同样采用MIT开源协议，“x”意味着Cross，表示交叉的意思，使用Cocos2d-x开发出来的游戏被允许编译和运行在多平台上。开发者只需使用C++语言编写一次游戏逻辑，便可将游戏运行在iOS、Android、Mac OS X以及Windows等平台上。毫无疑问，Cocos2d-x开启了Cocos2D引擎跨平台开发的时代。

同一时期，美国人采用C#，基于Mono改写成CocosNet，新西兰人Ryan Williams用JavaScript改写了HTML5 Canvas的版本等，社区也出现了Texture Packer、Glyph Designer等丰富的工具和编辑器，Cocos2D家族进入了鼎盛时期。

2011年，Cocos2D家族有了新的发展，集成式的编辑器开始涌现，其中包括CocoShop、CocosBuilder、Sprite Helper、Level Helper等。

2011年5月，Zynga公司雇佣了Cocos2d-iPhone的两位核心作者Ricardo Quesada和Rolando Abarca，两位程序员分别从阿根廷和智利迁移到Zynga的旧金山总部工作。

2011年年底，谷歌赞助Cocos2d-x团队将Cocos2d-x移植到Cocos2d-HTML5版本，实现Web游戏类型的覆盖，特别是移动Web游戏的开发。

2012年1月，林顺负责维护的Cocos2d-HTML5项目正式启动，并于同年8月发布了第一个稳定版本v2.0。因为它是基于Cocos2d-x 2.0版本移植的，所以Cocos2d-HTML5第一版发布的就是2.0版本，并没有发布v1.0的相关版本。与此同时，Rolando Abarca也主导并开发了一套基于Cocos2d-x和SpiderMonkey的JavaScript自动绑定技术（详见第16章）。

2012年12月4日，Cocos2d-x团队发布了第一个Cocos2d-HTML5联合版本，从那时起，Cocos2d-HTML5的游戏便可同步发布到Web和原生游戏平台上。

2013年11月，Cocos2d之父Ricardo Quesada加入Cocos2d-x团队，并出任Cocos2d-x团队的首席架构师。

2014年3月，为了提供更加一致的开发体验，真正实现跨全平台，Cocos2d-x团队将Cocos2d-x JSB模块从Cocos2d-x中独立出来，并对Cocos2d-HTML5进行整合升级，将Cocos2d-HTML5改名为Cocos2d-JS，然后发布了Cocos2d-JS引擎。

2014年9月12日，Cocos2d-x团队发布了Cocos2d-JS的Cocos2d-JS v3.0 final稳定版本，Cocos2d-JS应用而生。

2015年7月21日，Cocos2d-x团队发布了Cocos2d-x 3.7版本，为了统一Cocos2d-x引擎，Cocos2d-JS被并入Cocos2d-x引擎中。

1.2 Cocos2d-JS 介绍

大千世界，奥妙无穷，我们穷一生仍不能学习其皮毛。因此，学习任何东西之前，都应该明确所学之物为何物，为何而学，学习Cocos2d-JS游戏引擎自然也不例外。官方对Cocos2d-JS的介绍是：

Cocos2d-JS 是跨全平台的游戏引擎，采用原生JavaScript语言，可发布到Web平台、iOS、Android、Windows Phone 8、Mac、Windows等平台。该引擎基于MIT开源协议，完全开源、免费，易学易用，拥有活跃的社区支持。Cocos2d-JS让2D的游戏编程门槛更低，使用更加容易和高效。和其他类似游戏框架相比，它定义了更加清晰的2D游戏编程的基本组件，采用易学易用的API设计，并采用全球领先、具备原生性能的脚本绑定解决方案实现游戏的跨原生平台发布，开发效率更高，使用上最简单。

那么，Cocos2d-JS和Coco2d-x是什么关系呢？实际上，Cocos2d-JS是Cocos2d-x的JavaScript版本。Cocos2d-JS的前身为Cocos2d-HTML5，在3.0版本之后，官方对Web引擎Cocos2d-HTML5和Native引擎Cocos2d-x进行整合，并为Web和各原生平台开发提供了一套统一的工作流，开发者只需要关注自己游戏的JavaScript业务逻辑代码，然后使用Cocos Console工具管理开发以及发布等流程。图1-4为Cocos2d-JS引擎的架构图。

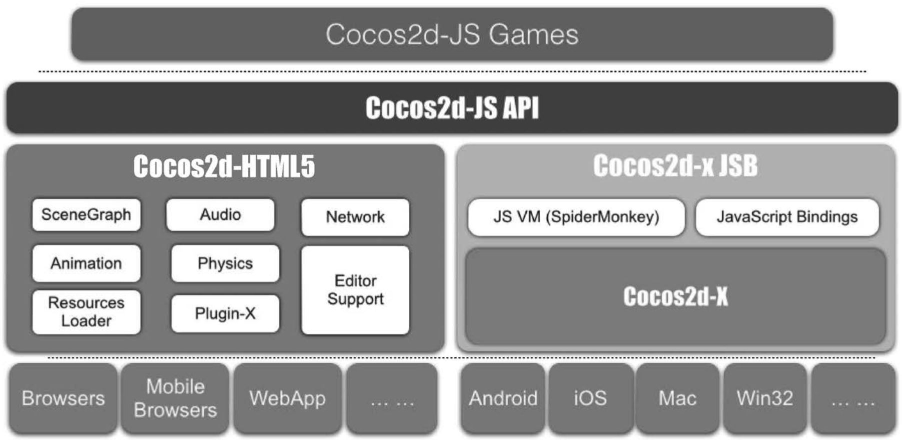


图1-4 Cocos2d-JS引擎的架构图

如图1-4所示，可将Cocos2d-JS引擎的结构分为4层。最底层为支持的平台，包括Web、iOS、Android、Mac OS X以及Windows等。

中间层为Cocos2d-HTML5和Cocos2d-x JSB（JavaScript Binding的缩写）这两个引擎。其中，Cocos2d-HTML5为一个采用JavaScript语言编写的独立引擎，采用Canvas或者WebGL渲染，并且完全兼容HTML5规范，使用Cocos2d-HTML5编写的游戏可以运行在浏览器上。

Cocos2d-x JSB则是一个介于Cocos2d-x原生代码和JavaScript代码之间的桥接层。Cocos2d-x JSB实现了JavaScript代码和Cocos2d-x引擎之间的相互调用，而这一实现得益于SpiderMonkey。SpiderMonkey是一个JavaScript虚拟机，采用C/C++开发，由Mozilla维护。SpiderMonkey虚拟机不仅可以被嵌入在浏览器上，也可被嵌入到任何C++程序中。在Cocos2d-x JSB中，开发者编写的JavaScript游戏业务逻辑代码就是SpiderMonkey负责解析和运行的。值得一提的是，Cocos2d-JS中嵌入的SpiderMonkey是被Cocos2d-x团队改造过的，从而支持了Cocos2d-x的类型、数据结构以及对象等。

综上所述，使用Cocos2d-JS开发出来的游戏不仅可以运行在浏览器上，还可以通过Cocos2d-x JSB的支持将游戏打包到原生平台上，例如iOS、Android以及Mac OS X等。

再上面一层为Cocos2d-JS API层，它保证了Cocos2d-HTML5和Cocos2d-x JSB的API高度一致，从而让开发者开发出来的游戏不用修改代码或者只修改少量代码，就可以打包为原生平台游戏，从而实现一次编码、全平台运行的畅爽体验。最上层即为JavaScript编写的Cocos2d-JS游戏逻辑代码。

1.3　引擎目录结构

从Cocos2d-x 3.7版本之后，官方将Cocos2d-JS引擎整合到Cocos2d-x中。Cocos2d-x引擎可在Cocos官网下载，其下载地址为：http://www.cocos.com/download/。当然，亦可从Cocos2d-x的GitHub仓库拉取，仓库地址：https://github.com/cocos2d/cocos2d-x。下载完成后，引擎包的主要内容如表1-1所示。

表1-1　引擎包的主要内容

目录或文件名	内容简介
AUTHORS	作者目录，包含所有给Cocos2d-x引擎贡献代码的开发者
build	包含测试例子、cocos2d_lib的Xcode以及Visual Studio工程
CHANGELOG	所有历史版本详细改动列表
CMakeLists.txt	cmake配置文件
cocos	Cocos2d-x引擎源代码
CONTRIBUTING.md	贡献代码指南
docs	包含JavaScript代码风格规范、当前发布说明和当前版本升级指南
download-deps.py	下载第三方库的脚本
extensions	第三方扩展
external	存放第三方库的文件夹
licenses	所有许可协议
plugin	插件
README.cmake	针对cmake用法的说明文件
README.md	Cocos2d-x引擎简介

（续）

目录或文件名	内容简介
setup.py	Cocos Console的安装脚本
templates	Cocos Console创建项目时使用的模板
tests	各分支的测试项目
tools	工具文件夹
—bindings-generator	脚本绑定工具
—cocos2d-console	Cocos Console工具
—tojs	JSB自动绑定配置文件以及生成脚本
—tolua	Lua绑定配置文件以及生成脚本
web	Cocos2d-JS游戏引擎

1.4　小结

通过本章的学习，我们知道了Cocos引擎的家族史，了解了Ricardo Quesada为Cocos2D之父，王哲和林顺分别为Cocos2d-x和Cocos2d-HTML5引擎的创始人。除此之外，我们还了解了Cocos2d-x和Cocos2d-JS之间的关系，以及Cocos2d-JS的引擎架构。最后，我们还介绍了Cocos2d-x引擎包的目录结构。

1.5　参考资源

本章的参考资源如下。

- 维基百科Cocos2d：https://en.wikipedia.org/wiki/Cocos2d。
- Cocos官网：http://www.cocos.com/。
- LiquidFun流体引擎：http://google.github.io/liquidfun/。
- 傅思杰（偶尔e网事）整理的Cocos资料大全：https://github.com/fusijie/Cocos-Resoure。

第2章 Hello World

古语有云："工欲善其事，必先利其器。"唯有配备好了装备，方可上阵杀敌。学习Cocos2d-JS游戏引擎，亦是如此。本章将讲解Cocos2d-JS在各平台下的环境搭建、Hello World解析以及Cocos DevTools工具等。

本章内容：

- Cocos Console
- 创建、编译和运行工程
- HelloWorld的目录结构
- 项目在Web和Native上的启动流程
- 项目在各平台下的打包以及部署
- js-tests测试工程
- 为《保卫萝卜2》项目做准备
- 实例——利用Cocos DevTools学习Cocos2d-JS

2.1 Cocos Console

Cocos Console是Cocos2d-x引擎下的一个命令行工具，它用来管理Cocos工程，其中包含创建、运行、编译、调试以及打包项目等。

Cocos Console位于引擎包cocos2d-x/tools/cocos2d-console目录下，通过运行引擎包目录下的setup.py脚本即可安装。在安装的过程中，Cocos Console需要开发者提供Android NDK、Android SDK和Apache ANT的文件路径。另外，Cocos Console是一个采用Python语言编写的跨平台脚本工具，所以在安装Cocos Console之前，需要先安装好Python。

2.1.1 安装 Python

在Mac OS X中，操作系统本身自带了Python，而在Windows操作系统中，Python则需要我们自行下载并安装，其下载地址为：https://www.python.org/downloads/index.html。若你的Mac OS X

系统中没有Python，也可通过此地址下载安装。下载至Mac OS X和Windows上的安装包分别是一个.pkg或者.msi文件，例如我下载的是python-2.7.10-macosx10.6.pkg和python-2.7.10.msi文件，如图2-1所示。

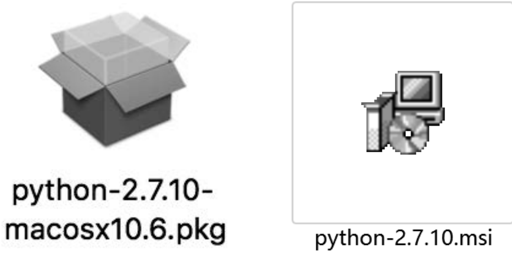


图2-1　Python在Mac OS X（左）和Windows上的安装包（右）

1. 在Mac OS X 上安装Python

在Mac OS X上，Python的安装较为简单，双击打开python-2.7.10-macosx10.6.pkg文件，然后一直点击“继续”按钮，即可完成安装。

2. 在Windows上安装Python

在Windows操作系统中，Python的安装步骤可分如下两步。

(1) Python安装：双击python-2.7.10.msi文件将其打开，将Python安装在指定磁盘路径下，这里我将Python安装到C:\Python27。

(2) 环境变量配置：当Python安装成功后，还需要配置系统环境变量，具体步骤如下。

① 右击“我的电脑”，从打开的快捷菜单中选择“属性”菜单。

② 此时会打开“属性”对话框，从中选择“高级系统设置”选项卡。

③ 点击“环境变量(N)...”按钮，打开“环境变量”对话框。

④ 从“系统变量(S)”列表框中选择“ path”，然后点击 “编辑(I)...” 按钮。

⑤ 在变量值的最后输入Python的安装路径，并在路径前面加上一个分号，例如;C:\Python27，这是因为每个环境变量之间需要用分号隔开。

⑥ 点击“确定”按钮关闭窗口。至此，Windows上的Python安装成功。

3. Python验证

在Mac OS X下，可以打开终端（Windows中为命令行程序），输入`python --version`。注意，`version`之前为两个杠（-），若出现如图2-2或图2-3所示的界面，便说明Python安装成功。

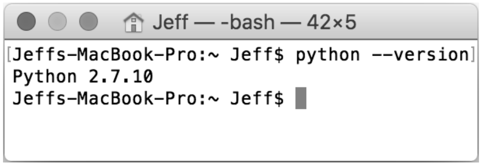


图2-2　Mac OS X下Python验证

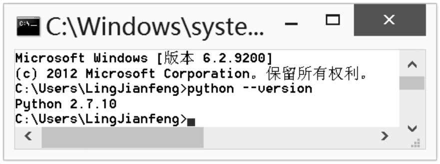


图2-3　Windows下Python验证

说明　本书撰写时，setup.py脚本采用Python 2.7编写，并不支持Python 3，所以本书在Mac OS X上采用Python 2.7.10的64位版本。在Windows上，选用Python 2.7.10的32位版本。

2.1.2　Android 环境配置

当安装好Python之后，你便可以开始准备Android相关的软件包了。当然，若你不需要支持Android，除了Apache Ant之外，其余步骤可以跳过，不必配置。Android所需的相关软件包如下所示。

- **Apache Ant**：将软件编译、测试、部署等步骤联系在一起加以自动化的一个工具，大多用于Java环境中的软件开发。下载地址：http://ant.apache.org/bindownload.cgi。
- **Android SDK**：即Software Development Kit的简称，中文译为软件开发工具包。在Android中，它为开发者提供了库文件以及其他开发所用到的工具。下载地址：http://developer.android.com/ tools/sdk/ndk/index.html。
- **Android NDK**：即Native Development Kit的简称，它是一系列工具的集合，可以帮助开发者快速开发C/C++的动态库。另外，它还能自动将.so文件和Java应用一起打包成.apk。下载地址：https://developer.android.com/sdk/index.html?hl=sk。
- **JDK**：Java的开发工具包，包括Java运行环境、Java工具和Java基础类库。下载地址：https://www.oracle.com/downloads/index.html 。

当下载好以上所需要的依赖后，便可以正式安装Cocos Console了。此时打开终端（Windows下为命令行程序），进入Cocos2d-x引擎目录下，然后再运行setup.py脚本，相关命令如下：

```
1  cd /Users/Jeff/Documents/Cocos/cocos2d-x
2  python setup.py
```

其中，Jeff为笔者的用户名。在读者的电脑上，以读者的用户名为准。然后根据提示，将Cocos Console所需的文件路径拖曳进去，最后根据末尾行提示进行对应的操作。例如，在Mac OS X上应该再运行如下命令：

```
1  source /Users/Jeff/.bash_profile
```

而在Windows操作系统中，则重启命令行程序即可。但是实际上，在Mac OS X上，也需要重新启动终端程序。另外，值得注意的是，Apache Ant路径需要指定到bin文件夹下。Mac OS X和Windows操作系统上Cocos Console的安装过程分别如图2-4和图2-5所示。

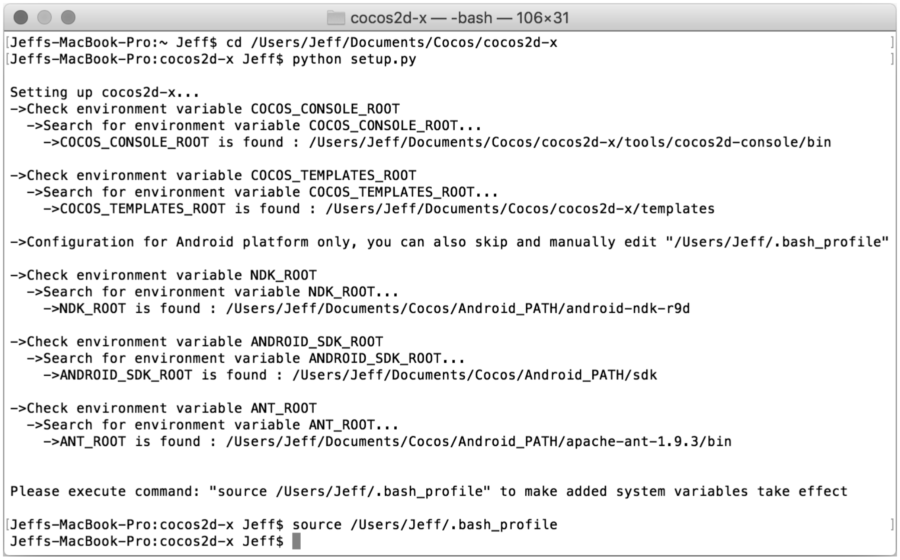


图2-4 Mac OS X下Cocos Console的安装过程

注意 在Mac OS X上将文件夹拖曳进去时，路径的末尾会有一个空格，应将空格删除。



图2-5 Windows下Cocos Console的安装过程

此时，Cocos Console安装成功。若要卸载Cocos Console，则Mac OS X用户可删除/Users/用户名/下.bash_profile文件中对应的值，而Windows用户只需删除对应的系统环境变量值即可。

2.2　创建、编译和运行工程

当Cocos Console安装完毕后，便可通过Cocos Console创建Cocos2d-JS工程，具体有如下几种方式：

```
1  // 创建一个名为projectName，并同时包含Cocos2d-HTML5和Cocos2d-x JSB的项目
2  cocos new projectName -l js
3  // 创建一个名为projectName，且仅含Cocos2d-HTML5的项目， --no-native表示不需要支持Native平
      台（iOS、Android、Mac、Windows等），仅支持浏览器即可
4  cocos new projectName -l js --no-native
5  // 在桌面上创建一个名为projectName的项目
6  cocos new projectName -l js -d ./Desktop
7  // 在桌面上创建一个名为projectName的项目，并设置为竖屏
8  cocos new projectName -l js -d ./Desktop --portrait
```

此外，亦可通过如下命令在桌面上创建一个名为HelloWorld的工程：

```
1  cd ./Desktop
2  cocos new HelloWorld -l js
```

其中，`-l`表示采用的语言，可选值为`cpp`、`lua`以及`js`。运行命令之后，便可在桌面上看到HelloWorld文件夹。图2-6为HelloWorld工程的创建过程。

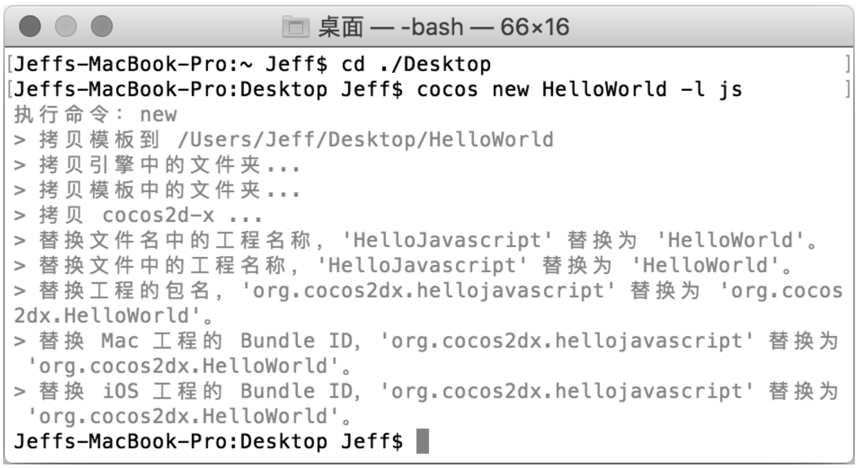


图2-6　Cocos Console创建工程的过程图

当项目创建完毕后，可以通过下列命令将项目运行在浏览器中，效果如图2-7所示：

```
1  cd ./Desktop/HelloWorld
2  cocos run -p web
```

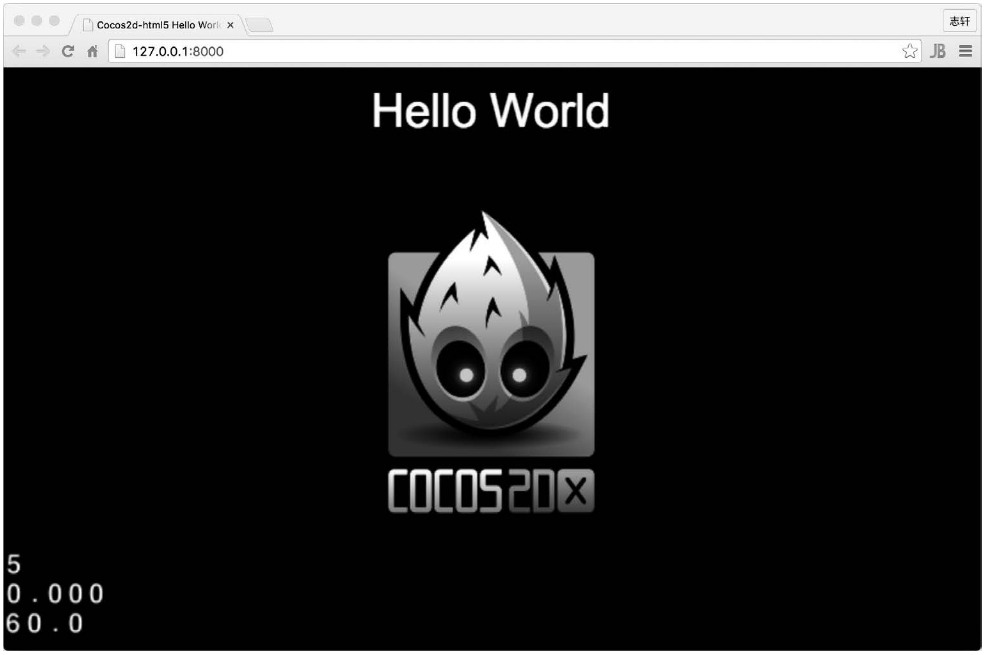


图2-7　运行在浏览器上的HelloWorld工程

除创建命令外，Cocos Console还为工程提供了运行、编译等命令，具体如下：

```
1  //运行在指定的平台上
2  cocos run -p web|ios|android|mac|win32
3  //将项目工程打包到指定的平台上
4  cocos compile -p web|ios|android|mac|win32 -m release
```

值得说明的是，若发布在Web平台上，可添加`--advanced`参数，它表示使用Closure编译器的高级模式编译JS文件，这将获得更高的压缩率，但是有出现bug的风险，请读者自行斟酌使用。若出现bug，可参考https://developers.google.com/closure/compiler/docs/api-tutorial3解决。另外，Cocos Console提供了相关的`help`指令，方便开发者查询Cocos Console相关的指令。下面举几个`help`指令的例子，其中`help`可用字母`h`替代：

```
1  cocos new --help
2  cocos run --help
3  cocos compile --help
```

2.3　HelloWorld 的目录结构

此时，虽然基于Cocos2d-JS的HelloWorld工程创建完毕，并且也成功地运行在浏览器上，但是读者对HelloWorld工程可能还比较陌生，并不知道每个文件夹和文件是做什么的，也不知道项目是如何启动的。接下来，我将带大家走进Cocos2d-JS的HelloWorld世界。

打开刚刚创建好的HelloWorld工程，可以看到其目录结构如图2-8所示。

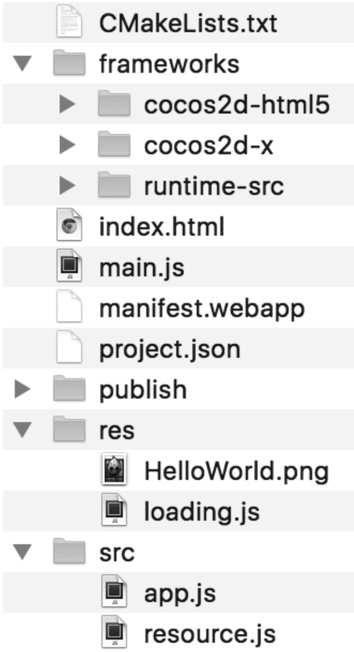


图2-8　HelloWorld目录结构

图2-8中有4个一级文件夹目录，一个.html文件、一个.json文件以及多个.js脚本等。表2-1为目录结构的详细介绍。

表2-1　目录结构

目录或文件名	内容简介
CMakeLists.txt	Cocos Console编译时所依赖的文件
frameworks	包含Web引擎和Native引擎
—cocos2d-html5	Cocos2d-HTML5游戏引擎
—cocos2d-x	Cocos2d-x游戏引擎
—runtime-src	各平台的项目工程文件，包含iOS、Mac OS X、Android以及Windows等
—index.html	Web工程的主页面，其主要内容和职责如下。 ❑ 包含用于显示游戏场景的canvas元素。 ❑ 引入用于引擎初始化和加载的引擎脚本：CCBoot.js。 ❑ 引入游戏启动的入口脚本：main.js。 ❑ 包含一些适配移动端浏览器页面的meta元素
—main.js	游戏入口文件，其中包含游戏初始化代码以及启动代码

（续）

目录或文件名	内容简介
—manifest.webapp	热更新配置文件
—project.json	工程配置文件
publish	该目录初始状态下不存在，当工程以发布模式打包后，会创建该文件夹并保存对应平台的发布包
Runtime	该目录初始状态下不存在，用来存储调试模式打包的工程执行文件
res	项目资源文件夹，用来存储所有图片、音频、字体以及动画等资源
—HelloWorld.png	默认资源图片
—loading.js	浏览器上页面启动的loading效果
src	项目脚本文件夹，用来存储游戏的所有JavaScript代码
—app.js	项目代码
—resource.js	资源的全局变量定义

其中，project.json为项目配置文件，其内容如下所示：

```
1  {
2      "project_type": "javascript",
3      "debugMode" : 1,
4      "showFPS" : true,
5      "frameRate" : 60,
       "noCache" : false,
6      "id" : "gameCanvas",
7      "renderMode" : 0,
8      "engineDir":"frameworks/cocos2d-html5",
9      "modules" : ["cocos2d"],
10     "jsList" : [
11         "src/resource.js",
12         "src/app.js"
13     ]
14 }
```

这些属性影响着整个Cocos2d-JS项目，一些属性的背后更是深藏功与名。各属性的意义如下。

- ❑ **project_type**：项目类型。
- ❑ **debugMode**：表示程序的调试级别，默认值是1，可选值为0到6。
  - 0：不显示任何调试信息。
  - 1：`cc.error`、`cc.assert`、`cc.warn`、`cc.log`在调试终端中打印信息，这是默认值。
  - 2：`cc.error`、`cc.assert`、`cc.warn`在调试终端中打印信息。
  - 3：`cc.error`、`cc.assert`在调试终端中打印信息。
  - 4：`cc.error`、`cc.assert`、`cc.warn`、`cc.log`在canvas上显示信息，这是Cocos2d-HTML5引擎独有的功能。这在微信开发上是一个非常有用的功能。
  - 5：`cc.error`、`cc.assert`、`cc.warn`在canvas上显示信息，这是Cocos2d-HTML5引擎独有的功能。

    - 6：`cc.error`、`cc.assert`在canvas上显示信息，这是Cocos2d-HTML5引擎独有的功能。
- **showFPS**：若其值为`true`，则游戏窗口左下角会显示绘制函数调用次数、渲染时间以及帧率。默认取值为`true`，项目打包上线时，应将其设置为`false`。
- **frameRate**：设置期望帧率。游戏中的实际帧率会取决于游戏每帧消耗时间和运行平台等条件，期望帧率会保证游戏运行帧率不会超过期望帧率，并尽力运行在期望帧率上。
- **noCache**：设置是否让浏览器缓存html页面。
- **id**：Web引擎页面中`canvas`元素的`id`，仅服务于Cocos2d-HTML5引擎。
- **renderMode**：引擎绘制模式，仅服务于Cocos2d-HTML5引擎，在Native上无效，其值的可取范围为0~2，表示的意义如下。
    - 0：由引擎自动选择绘制模式，遵循“择优选择”的理念，即若支持WebGL，则优先考虑使用WebGL绘制，否则采用canvas绘制。
    - 1：强制使用canvas绘制模式。
    - 2：强制使用WebGL绘制模式，但是实际上WebGL仍然可能会在一些移动浏览器上被忽略而自动使用canvas绘制模式。
- **engineDir**：Cocos2d-HTML5引擎路径，仅在debug模式下有效，保持默认值即可。如果采用Cocos2d-JS Lite版开发游戏，则此字段可以忽略。
- **modules**：模块配置。可将游戏中需要引入的模块添加到此数组中，例如游戏中需要Chipmunk物理引擎支持，则应该在此数组中添加`chipmunk`字符串。此配置仅服务于Cocos2d-HTML5引擎，当Cocos Console在发布模式下编译生成项目时，会根据`modules`中所包含的模块进行打包。所以，`modules`应当保持精简，按需取值，避免打出的包中引入了没有用到的模块，增大了游戏脚本的大小。关于存在哪些模块以及每个模块的定义，可以参考frameworks/cocos2d-html5/modulesConfig. json文件。
- **jsList**：开发者的JS脚本游戏逻辑代码列表，游戏中所依赖的脚本都应该放入这个列表中。此外，应当注意这些JS文件的相互依赖关系以及加入的先后顺序。

2.4　项目在 Web 和 Native 上的启动流程

Cocos2d-JS是一个跨浏览器和原生平台的游戏引擎，在对应平台上的启动流程自然也有所不同。

1. Web

在Web上，index.html生成canvas，并加载CCBoot.js文件，然后CCBoot.js读取project.json文件，从而预加载resource.js以及jsList中所有的.js文件。到此，游戏中所有的代码文件被加载完毕。随后，index.html又引入main.js文件，最终main.js启动了游戏。整个过程的代码如下，启动流程图如图2-9所示：

```
1  <body style="padding:0; margin: 0; background: #000;">
2  <script src="res/loading.js"></script>
```

```
3  <canvas id="gameCanvas" width="800" height="450"></canvas>
4  <script src="frameworks/cocos2d-html5/CCBoot.js"></script>
5  <script cocos src="main.js"></script>
```

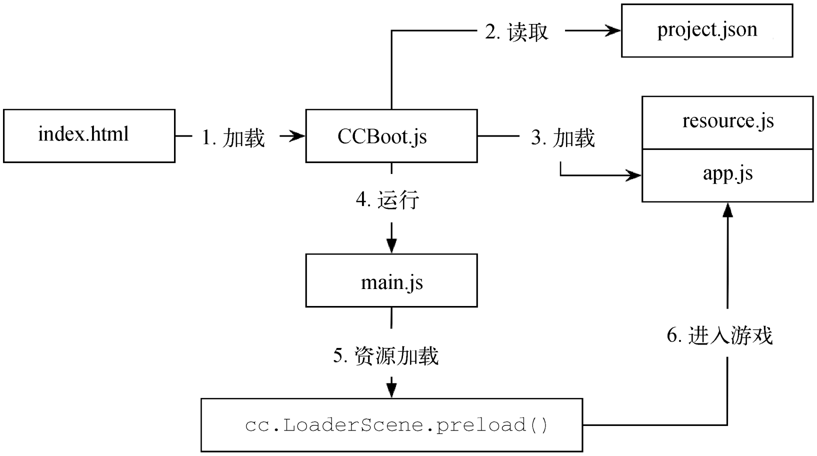



图2-9　Cocos2d-JS游戏在浏览器上的启动流程

2. Native

在Native中，Cocos2d-JS工程的启动流程相对简单一些。打开HelloWorld/frameworks/runtime-src/Classes/AppDelegate.cpp文件，可以在AppDelegate类的applicationDidFinishLaunching函数中看到如下代码：

```
1   bool AppDelegate::applicationDidFinishLaunching(){
2       // 导演初始化
3       auto director = Director::getInstance();
4       auto glview = director->getOpenGLView();
5       if(!glview) {
6           #if(CC_TARGET_PLATFORM == CC_PLATFORM_WP8) || (CC_TARGET_PLATFORM ==
            CC_PLATFORM_WINRT)
7               glview = cocos2d::GLViewImpl::create("HelloWorld");
8       #else
9           glview = cocos2d::GLViewImpl::createWithRect("HelloWorld",
            Rect(0,0,900,640));
10      #endif
11          director->setOpenGLView(glview);
12      }
13      director->setAnimationInterval(1.0 / 60); // 帧率设置
14      // 获取脚本引擎（SpiderMonkey）单例对象
15      ScriptingCore* sc = ScriptingCore::getInstance();
16      sc->addRegisterCallback(register_all_cocos2dx);
17      sc->addRegisterCallback(register_cocos2dx_js_core);
18      sc->addRegisterCallback(jsb_register_system);
19      // ......
20      sc->start();
```

```
21     sc->runScript("script/jsb_boot.js"); // 运行jsb_boot.js脚本
22     // ......
23     ScriptEngineProtocol *engine = ScriptingCore::getInstance();
24     ScriptEngineManager::getInstance()->setScriptEngine(engine);
25     ScriptingCore::getInstance()->runScript("main.js");
26     return true;
27 }
```

在第21行代码中，脚本引擎运行jsb_boot.js脚本，此脚本中同样读取project.json配置文件，然后在第25行中运行main.js文件，从而进入游戏。

说明　Native即为原生操作系统平台，例如iOS、Android、Mac OS X等。总之，在Cocos2d-JS中，除了浏览器外，都可以称为Native。

2.5　项目在各平台下的打包以及部署

在游戏开发的过程中，可能需要和Objective-C或者Java语言交互（参见第13章），这就需要你在对应的开发工具中打开Cocos2d-JS工程。前面我提到过frameworks/runtime-src文件夹保存着各平台的工程文件，接下来就来看看应当如何使用这些工程文件。

1. iOS和Mac OS X

iOS和Mac OS X上的工程文件用起来比较简单，在安装了Xcode的前提下，双击打开frameworks/runtime-src/proj.ios_mac/HelloWorld.xcodeproj文件即可。值得说明的是，这个工程文件中集成了iOS和Mac OS X项目，你可以点击如图2-10所示的HelloWorld-mobile按钮选择你所要的平台。

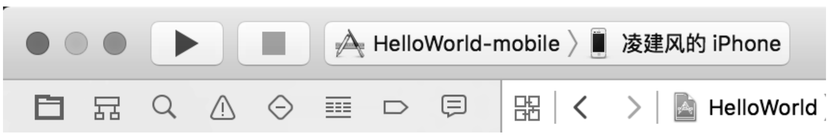


图2-10　iOS和Mac OS X工程选择

2. Android

在Android上，则需将HelloWorld/frameworks/runtime-src/proj.android导入到Eclipse中，操作步骤如图2-11、图2-12和图2-13所示。

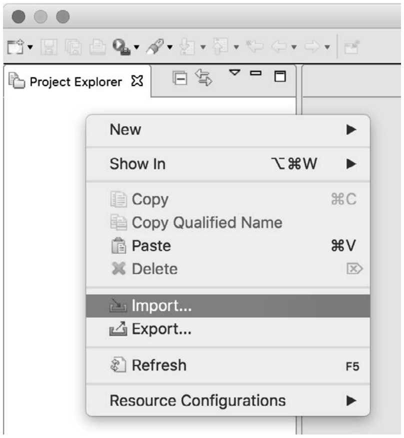


图2-11　导入工程

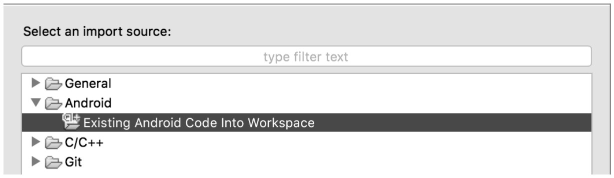


图2-12　选择Android工程

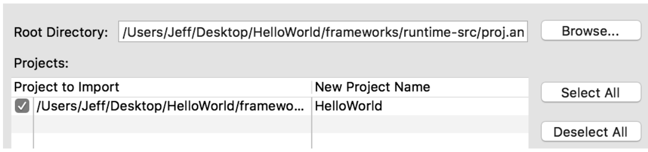


图2-13　选择Android工程路径

导入后，HelloWorld工程会报错，如图2-14所示，这是因为HelloWorld没找到libcocos2dx库项目，只需通过上述步骤再将HelloWorld/frameworks/js-bindings/cocos2d-x/cocos/platform/android/java也导入到 Eclipse中即可，完成后的效果如图2-15所示。

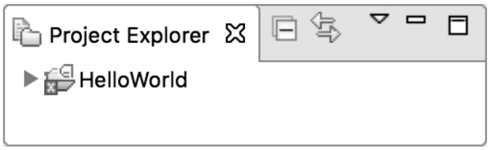


图2-14　将HelloWorld工程导入Eclipse

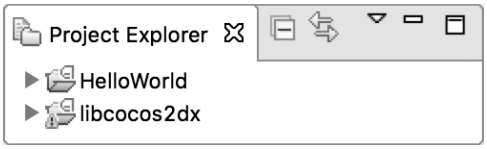


图2-15　将libcocos2dx库导入到Eclipse

3. Web部署

根据读者的需求，Web项目的部署也可能完全不同。这里我们以模拟局域网为例，讲解如何使用XAMPP在局域网上部署Cocos2d-JS Web项目，这样可以方便读者在开发过程中通过真机测试。

XAMPP是一款集成了Cross-Platform (X) 、Apache、MySQL、PHP以及Perl的软件，安装完之后，便可在个人电脑上搭建服务器环境。XAMPP是多平台软件，所以它提供了Mac OS X、Windows、Linux等操作系统。其安装步骤较为简单，下载完安装包之后，直接双击来打开它，然后一直点击Next按钮直到安装完毕即可。该软件的下载地址：https://www.apachefriends.org/download.html。

XAMPP在Mac OS X和Windows中的界面图分别如图2-16和图2-17所示。

说明 实际上，在终端（Windows中对应命令行程序）调用命令`python -m SimpleHTTPServer`，也可以直接快速搭建Web服务器，相关代码如下：

```
1 cocos new HelloJS -l js -d ./Desktop
2 cd ./Desktop/HelloJS/
3 python -m SimpleHTTPServer 8080
```

其中，8080为指定端口，可省略，默认值为8000。当执行上面第三行指令后，便可以在浏览器中通过http://localhost:8080/地址打开HelloJS项目。另外，在局域网内，localhost可用IP地址代替。

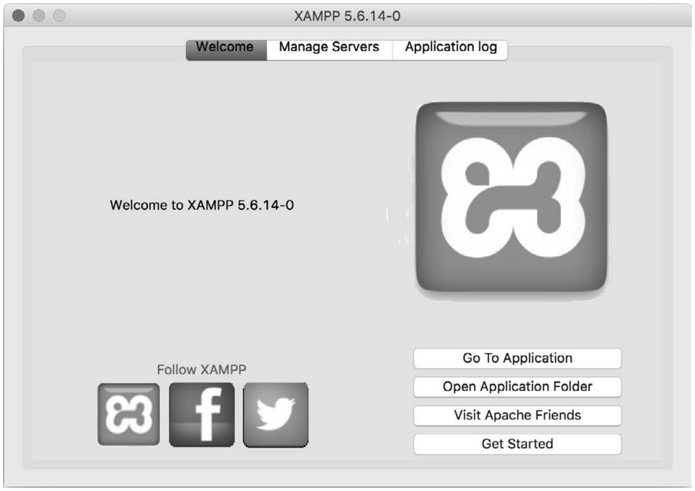


图2-16 XAMPP在Mac OS X下的界面图

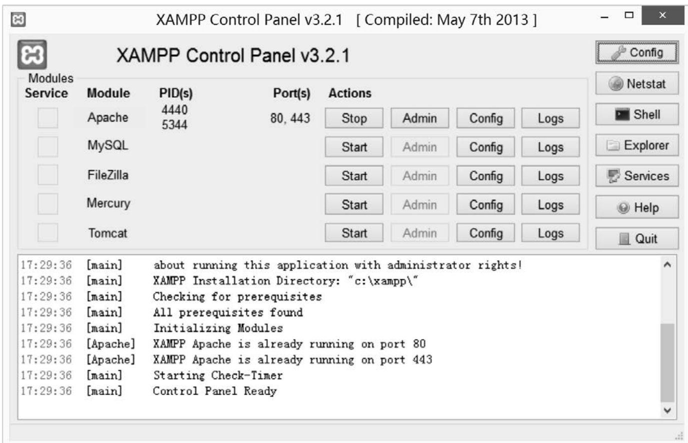


图2-17 XAMPP在Windows下的界面图

其中，在Mac OS X下可以通过点击Open Application Folder（Windows操作系统中对应Explorer）按钮，打开XAMPP安装路径。在此目录中，可以看到一个名为htdocs的文件夹（如图2-18所示），将打包好的Web项目放在此文件夹中，便可在局域网内通过IP地址访问到。当然，这需要先选中Manage Servers面板中的Apache Web Server选项，然后点击Start按钮（Windows下直接点击Start按钮即可，如图2-17所示），启动Apache Web服务器。

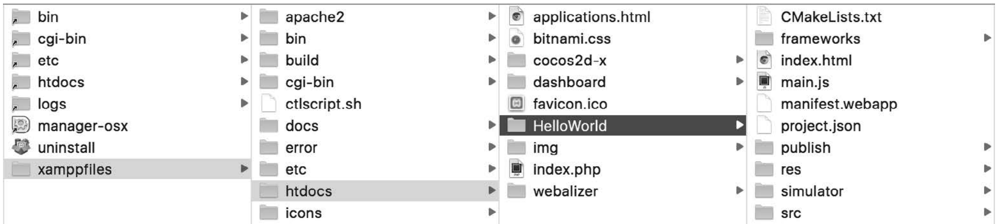


图2-18　htdocs目录结构

我习惯把工作区间设在XAMPP/htdocs路径下，如图2-18所示。因为这样在启动Apache Web的情况下，便可在浏览器通过http://localhost/HelloWorld/地址直接访问工程。当然，也可以通过IP地址的方式访问，如图2-19所示。

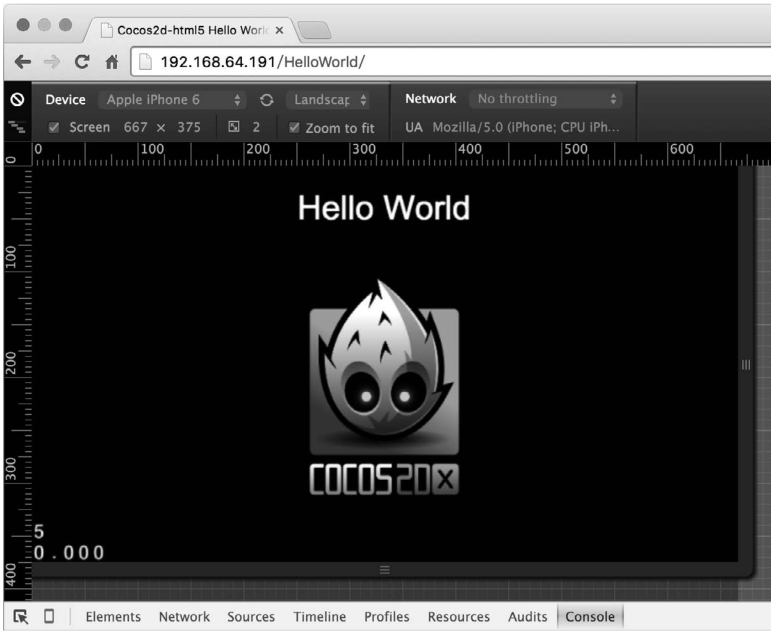


图2-19　通过IP地址访问工程

另外，值得说明的是，当Web项目打包之后，HelloWorld文件夹下将出现publish文件夹，而publish文件夹又包含html5文件夹（如图2-20所示），此文件夹便是发布之后的项目，它通过终端

命令`cocos compile -p web -m release`得到。

此时读者可将自己的移动设备开启WiFi，保证和电脑在同一局域网下，然后在移动设备的浏览器上键入http://192.168.64.191/HelloWorld/publish/html5/，也可将此地址通过网络第三方工具转为二维码，然后通过扫描二维码在移动设备上访问HelloWorld项目，从而起到一个Web项目部署模拟的效果。

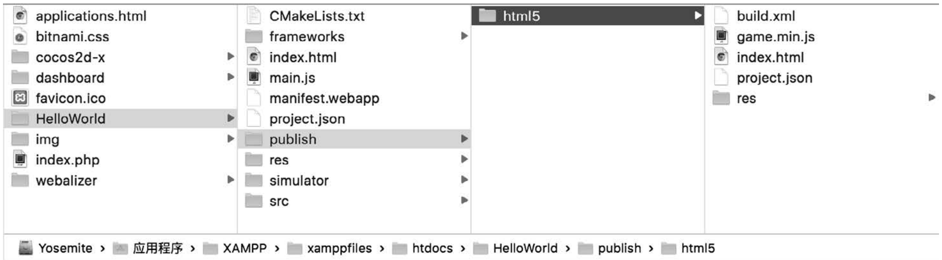


图2-20　打包后的html5游戏包

2.6　js-tests 测试工程

js-tests位于cocos2d-x/tests/文件夹下，它是Cocos2d-JS引擎的测试例子，也是一个非常好的学习资料，我强烈建议你经常打开这个工程看看，多阅读它的实现代码。并且，在实际开发中，js-tests也是一个非常好的参考示例。js-tests项目工程运行在浏览器上的效果如图2-21所示。

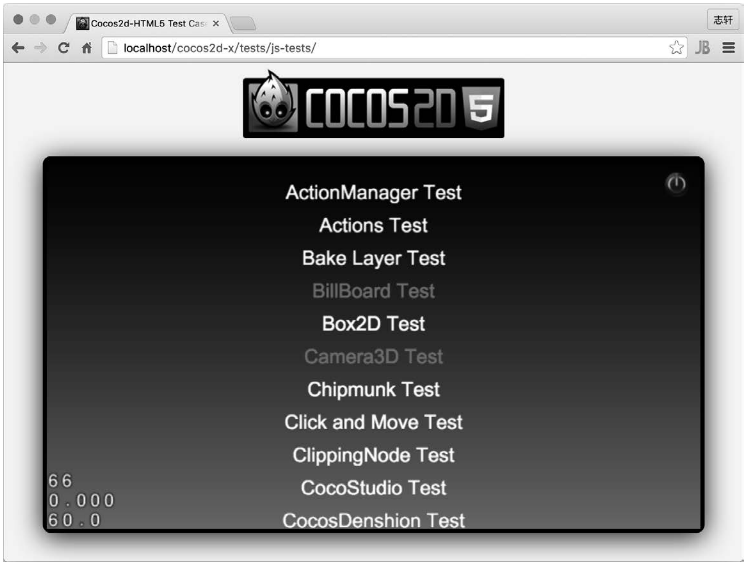


图2-21　js-tests在浏览器上的运行效果

注意 1. cocos2d-x/tests/js-tests/index.html不可以通过双击直接打开运行，它需要放在服务器上，即将cocos2d-x文件夹放在XAMPP/htdocs路径下，然后在浏览器中输入http://localhost/cocos2d-x/tests/js-tests/index.html，这样才能正确打开js-tests工程。

2. 若在浏览器上出现一些奇奇怪怪的问题，让读者百思不得其解，可能是浏览器缓存的问题，不妨考虑清理下浏览器缓存或在开发过程中将project.json文件中的`noCache`值设置为`true`。

2.7　为《保卫萝卜 2》项目做准备

经过前面的学习，读者应该对Cocos2d-JS有了初步的了解，并且此时读者的电脑上应该是安装好了Cocos Console. 那么从现在开始，我们将和读者一起来制我们自己的《保卫萝卜2》。

1.《保卫萝卜》简介

《保卫萝卜》是飞鱼科技于2012年7月推出的一款休闲塔防游戏，发布后迅速蹿升至App Store中国区榜首，并被超过120个国家推荐为热门游戏。其续作《保卫萝卜2：极地冒险》于2013年11月正式上线，发布当日仅6小时即登上中国区总榜第一，并持续位居榜首长达21天，是一款老少皆宜的国民级休闲塔防手游。《保卫萝卜》《保卫萝卜2》总下载量已接近两亿，《保卫萝卜3》由腾讯代理发行，将在今年上线。

《保卫萝卜2》游戏主页面、关卡选择页面、游戏玩法页面分别如图2-22、图2-23和图2-24所示。若读者没有玩过《保卫萝卜2》，我建议你去App Store或Google Play上下载一个，然后体验一番，这样对本书的学习将会有非常大的帮助。因为我们将《保卫萝卜2》这个项目贯穿到本书大部分章节中作为章节实例，让读者可以在学习枯燥无味的理论知识后，立马动手做出作品。



图2-22　《保卫萝卜2》游戏主页面

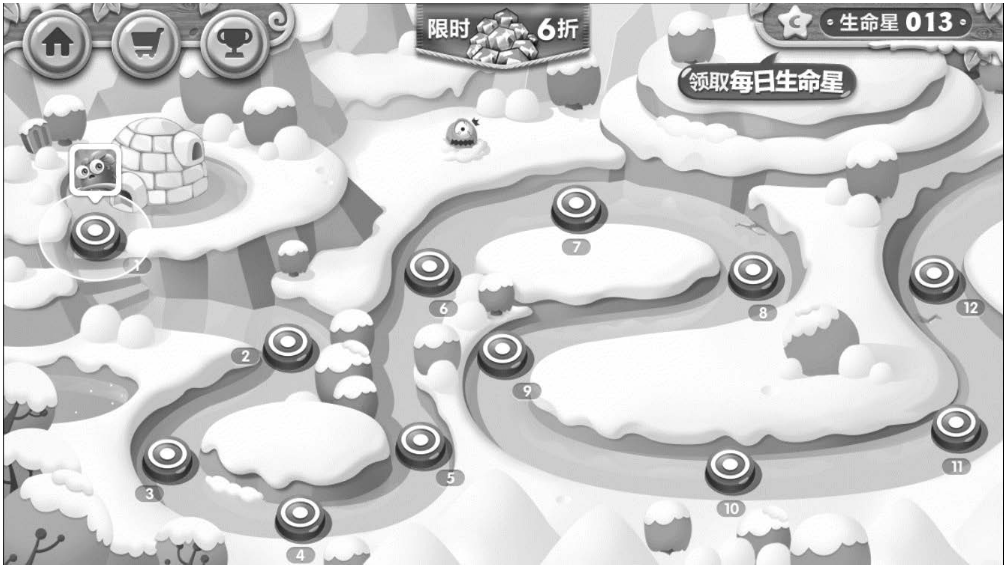


图2-23　《保卫萝卜2》关卡选择页面



图2-24　《保卫萝卜2》游戏玩法页面

说明　笔者虽已获得《保卫萝卜》系列素材官方授权，但因本书撰写时，《保卫萝卜3》还未上线，所以本书暂用《保卫萝卜2》的素材。

2. 使用WebStorm + Chrome开发《保卫萝卜2》

通过前面的学习，你已经知道了如何创建、编译、运行Cocos2d-JS游戏了，但是，你并不晓得通过何种工具开发Cocos2d-JS游戏，这里我推荐一套方案——WebStorm + Chrome。

WebStorm是JetBrains公司旗下一款JavaScript开发工具，被广大中国JavaScript开发者誉为"Web前端开发神器""最强大的HTML5编辑器""最智能的JavaScript IDE"等。WebStorm不仅提供了最智能的代码补全、代码格式化以及联想查询等高效功能，而且在代码高亮着色上也非常完美，其下载地址为：http://www.jetbrains.com/webstorm/。

Chrome是Google（谷歌）公司于2008年9月发布的多平台浏览器，它支持Mac OS X、Windows、iOS、Android以及Linux等操作系统。Chrome不仅调试方便，还提供了Mobile Emulation来进行真机模拟，这对Cocos2d-JS游戏的屏幕适配（详情可参见第7章）等是非常有帮助的。

当下载和安装好WebStorm和Chrome之后，启动终端（Windows下为命令行程序），通过如下命令在桌面上创建《保卫萝卜2》（这里将其命名为Carrot）工程：

```
1  cocos new Carrot -l js -d ./Desktop
```

接着，打开安装好的WebStorm，点击Create New Project，然后选择Empty Project，接着在Location中输入指定路径，最后点击Create按钮，如图2-25和图2-26所示。

说明 虽然官方推出Cocos IDE，但是我觉得Cocos IDE并不适合初学者使用，并且本书撰写时，Cocos IDE正在寻找新的解决方案，所以推荐WebStorm + Chrome的方式。若项目以Native为主，则可以考虑使用WebStorm或者Sublime编辑代码，然后在Xcode中运行项目。



图2-25 WebStorm主页面

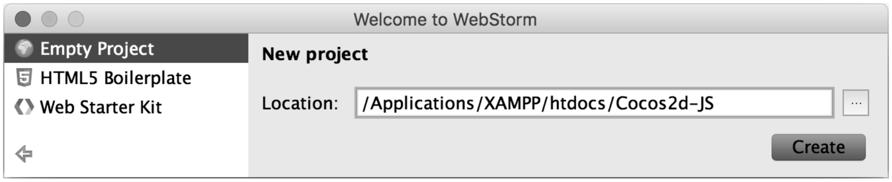


图2-26　创建工程

然后把刚才创建好的Carrot项目复制到XAMPP/htdocs/Cocos2d-JS文件夹下，此时回到WebStorm编辑器，右击Cocos2d-JS，选择Synchronize 'Cocos2d-JS'刷新目录，如图2-27所示，刷新后便可看到Carrot的目录结构了，如图2-28所示。

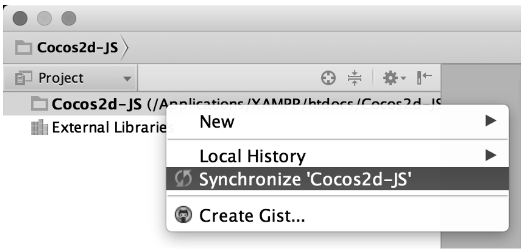


图2-27　刷新Cocos2d-JS目录

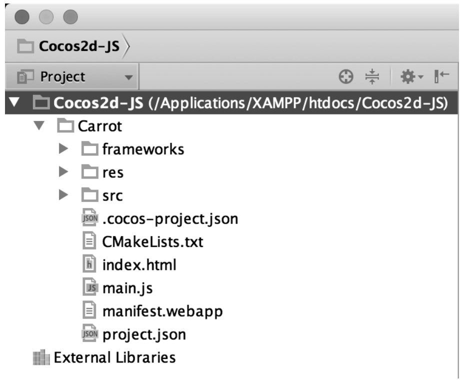


图2-28　Carrot目录结构

此时，你可以右击index.html，然后选择Debug 'index.html'，如图2-29所示，从而让Carrot运行在浏览器上。

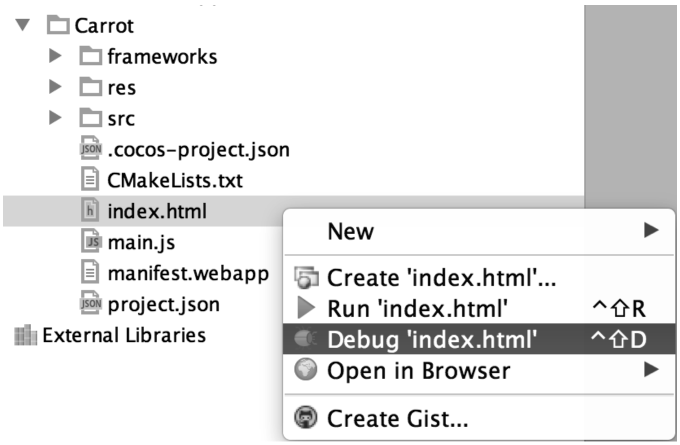


图2-29　选择Debug index.html

不幸的是，当你点击Debug 'index.html'之后，Chrome浏览器并没有反应，这是因为你还需要为Chrome安装一个名为JetBrains IDE Support的扩展程序（插件）。Chrome插件的安装步骤如下。

(1) 打开Chrome浏览器，点击如图2-30中箭头所指的菜单按钮。

(2) 选择“更多工具”→“扩展程序”。

(3) 点击底部“获取更多扩展程序”超文本链接，进入Chrome扩展程序商店，然后搜索安装JetBrains IDE Support插件。若搜索失败，可在Chrome打开扩展程序面板的情况下，将随书资源中的JetBrains_IDE_Support_2.0.8_0.crx文件拖到Chrome浏览器中直接安装。

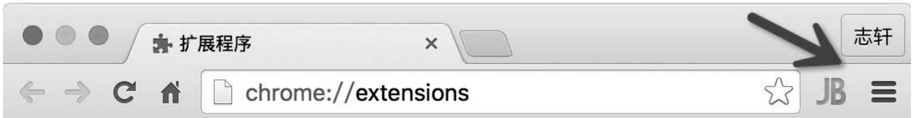


图2-30　点击Chrome菜单

安装成功后如图2-31所示。此时，再点击Debug 'index.html'，便可在浏览器中看到Carrot项目的运行效果。当然，此时你看到的仅仅只是一个HelloWorld的图片。不要急，在下一章的实例中，我们将带你开发《保卫萝卜2》的主页面，如图2-22所示。

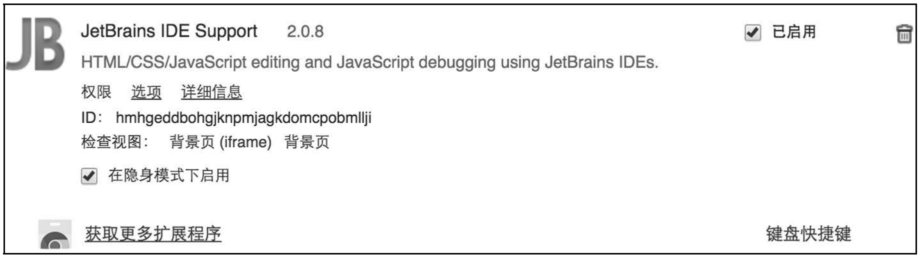


图2-31　JetBrains IDE Support插件

2.8　实例——利用 Cocos DevTools 学习 Cocos2d-JS

Cocos DevTools是Cocos2d-x团队于2015年2月26日推出的一个浏览器工具，可谓是神器，其下载地址为：http://h5.cocos.com/static/cocos-devtools/index.html。

Cocos DevTools工具的安装极其简单，只需设置浏览器显示书签栏，然后根据提示将按钮拖曳到书签栏中即可。安装完成后，在浏览器的书签栏中出现Cocos DevTools v1.0则表示安装成功，如图2-32所示。



图2-32　Cocos DevTools工具

安装好Cocos DevTools工具之后，在浏览器上运行一个Cocos2d-JS项目。例如，打开js-tests工程，然后点击书签栏中的Cocos DevTools工具，此时会出现如图2-33左下角所示的选项卡面板，其中包含场景元素和性能调试这两个选项卡。在场景元素中，你可以选中想要的元素，从而对它进行编程。例如在图2-33中，我选中了ActionManager Test元素，并将光标聚焦在地址栏中，再通过option + command + J（Windows按Ctrl + Shift + J键）组合键打开浏览器的Console（控制台），最后在Console中进行编码。图2-33中 Console 的代码如下：

```
1  var node = _cocos_devtools.curr;
2  var moveBy = cc.moveBy(2, cc.p(-200, 0));
3  node.runAction(moveBy);
```

第1行代码表示获取当前Cocos DevTools工具所选中的元素，第2行代码创建了一个动作（参见第4章），第3行代码运行这个动作。或许这些代码你现在看不太懂，但是并没有关系，因为这里抛出Cocos DevTools工具，只是希望你在阅读本书的时候，能够合理、高效地利用此工具。因为它能在很大程度上帮助你学习Cocos2d-JS引擎，即在你阅读本书的相关知识点时，可以立马通过此工具结合浏览器Console进行实践，从而降低了学习成本，提高了学习效率。

当然，我相信你已经感受到了，Cocos DevTools工具并非只是一个帮助学习的工具，它在项目的实际开发中是一个非常强大的助手。例如，当游戏里有非常多的元素时，你很难通过代码将想要的元素获取出来，但是通过Cocos DevTools工具，则容易许多，而这一般用在修复bug等方面上。

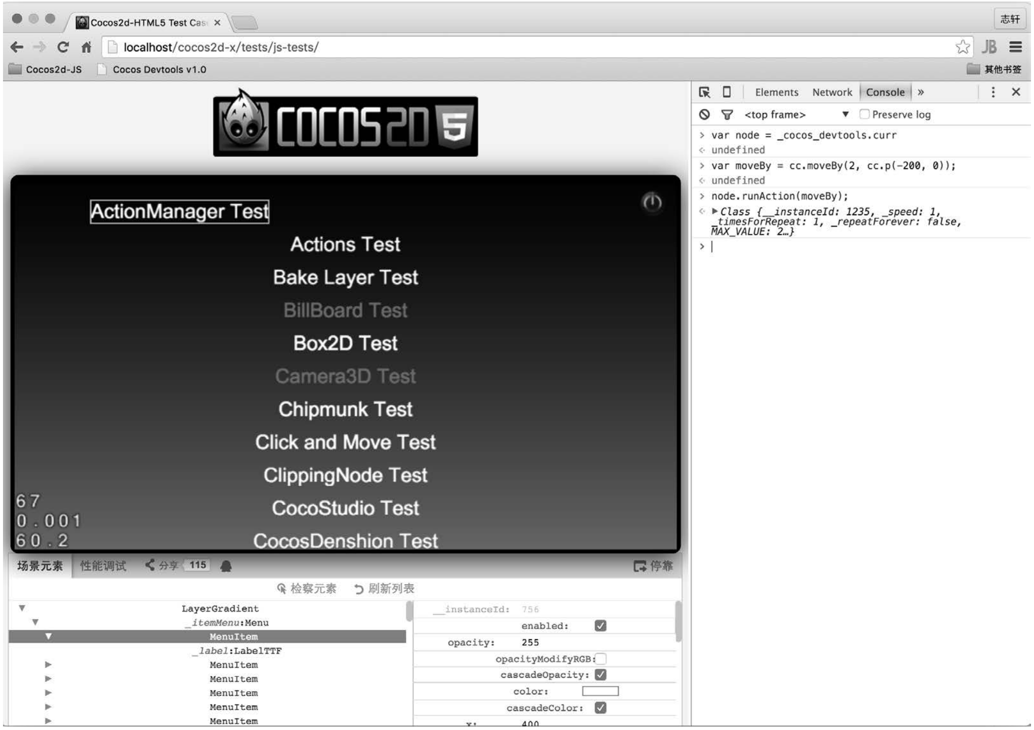


图2-33 使用Cocos DevTools工具学习Cocos2d-JS

2.9 小结

通过本章，我们知道了Cocos2d-JS项目的目录结构，了解每个文件夹和文件的作用，并且顺手搭建好了开发环境，掌握了Cocos2d-JS工程的创建、编译、运行、打包等技能。此外，还学到了通过WebStorm + Chrome的方式开发Cocos2d-JS游戏，并且掌握了如何使用Cocos DevTools工具。最后，创建了Carrot（《保卫萝卜2》）的项目工程。无奈笔者有点调皮且霸道，因为你并未在Carrot中看到任何和《保卫萝卜2》有关的东西。

2.10 参考资源

本章的参考资源如下。

- ❑ Cocos Console：https://github.com/cocos2d/cocos2d-console。
- ❑ Cocos2d-JS项目结构介绍：http://www.cocos2d-x.org/docs/manual/framework/cocos2d-js/4-essential- concepts/4-1-cocos2d-js-project/en。

第 3 章

核心框架

3

在我看来，游戏就像是一部可以实时交互的电影，玩家怎么操作，这部电影就怎么演。一部电影，我们时常关注这部电影的明星阵容，还有它的导演。在Cocos2d-JS中，同样也存在着这些角色。本章将围绕Cocos2d-JS游戏引擎的“明星阵容”展开，讲解这些角色的作用以及职责，它们是Cocos2d-JS游戏引擎最核心、最基础的部分。

本章内容：

- 导演（`cc.Director`）
- Cocos2d-JS坐标系
- 节点（`cc.Node`）
- 场景（`cc.Scene`）
- 层（`cc.Layer`）
- 精灵（`cc.Sprite`）
- 场景树
- 标签
- 实例——《保卫萝卜2》主页面开发

3.1 导演

在Cocos2d-JS中，导演（`cc.Director`）拥有至高无上的地位。如果把游戏比作一艘船，那么`cc.Director`就是这艘船的船长，它把控着整个游戏的运转和方向。例如，在游戏启动前，它负责设置游戏的运行环境；在游戏运行的过程中，它把控着整个游戏的主循环以及游戏场景的切换等工作。`cc.Director`是一个单例类，`cc.director`是`cc.Director`的单例对象，这意味着你可以在任何地方通过`cc.director`得到导演对象。或许现在我们可以跟随着`cc.director`，看看从游戏启动之前到游戏运行的整个过程中，`cc.director`都做了哪些事情。

3.1.1 设置帧率

帧率是用于测量显示帧数的量度，其单位为“每秒显示帧数”（Frame per Second，FPS）。一般情况下，电影以24fps播放，而在游戏中如果帧率低于30fps的话，游戏就会显得卡顿不连贯。在Cocos2d-JS中，帧率默认为60fps。若想修改默认帧率，可以直接修改工程下project.json文件中`frameRate`的值。

在游戏运行中，`cc.Director`也提供了动态操作帧率的API，你可以调用`cc.director.setAnimationInterval(fps)`方法来更改帧率。需要注意的是，这里的参数`fps`是帧率的倒数，表示游戏每秒绘制多少次，示例代码如下：

```
1  cc.director.setAnimationInterval(1.0 / 60);
```

说明 一般情况下，不建议修改引擎的帧率，但在某些特定的情况下，例如UI场景，你可以动态更改帧率，将帧率设置到30fps到40fps之间，使得游戏性能开销更少的同时又不会造成太大的影响。

3.1.2 初始化管理器

导演好比一艘船的船长，是整个游戏的统筹。既然是统筹，很多事情都不可能亲力亲为。很显然，它需要一些帮手来分担它的任务。这些帮手就是调度器、动作管理器和事件管理器。这些管理器，对应管理着自己的领域，它们都在`cc.Director`的`init()`方法中被初始化，相关代码如下：

```
1  cc.Director = cc.Class.extend({
2      init: function () {
3          // ......
4          this._scheduler = new cc.Scheduler();    // 创建[调度器]
5          if(cc.ActionManager){
6              this._actionManager = new cc.ActionManager(); // 创建[动作管理器]
7              this._scheduler.scheduleUpdate(this._actionManager,
               cc.Scheduler.PRIORITY_SYSTEM, false);
8          }else{
9              this._actionManager = null;
10         }
11         this._eventAfterDraw = new cc.EventCustom(cc.Director.EVENT_AFTER_DRAW);
           // 创建[事件管理器]
12         this._eventAfterDraw.setUserData(this);
13         this._eventAfterVisit = new cc.EventCustom(cc.Director.EVENT_AFTER_
           VISIT);
14         this._eventAfterVisit.setUserData(this);
15         this._eventAfterUpdate = new cc.EventCustom(cc.Director.EVENT_AFTER_
           UPDATE);
16         this._eventAfterUpdate.setUserData(this);
17         // ......
```

```
18            return true;
19        }
20 }
```

3.1.3　初始化渲染器

渲染器用于将图片或者3D模型素材绘制到屏幕上。因为Cocos2d-JS是一个跨全平台的游戏引擎，使用它开发出来的游戏可以运行在浏览器和本地上，所以，在各平台上采用的渲染器也是不一样的。

在HTML5中，iOS 8以上以及所有PC系统中，默认采用WebGL渲染，其他浏览器采用canvas绘制，例如Android手机上的浏览器。

在Native上，全部采用OpenGL ES 2.0渲染。

3.1.4　获取屏幕大小

在游戏开发中，时常需要获取当前屏幕的大小，然后参考屏幕的大小将游戏中的可视对象定位在屏幕上。cc.Director提供了几种不同概念的屏幕大小，相关代码如下：

```
1  cc.Director = cc.Class.extend({
2      getWinSize: function () {
3          return cc.size(this._winSizeInPoints);
4      },
5      getWinSizeInPixels: function () {
6          return cc.size(this._winSizeInPoints.width * this._contentScaleFactor,
           this._winSizeInPoints.height * this._contentScaleFactor);
7      },
8      getVisibleSize: null,
9      getVisibleOrigin: null,
10 });
```

其中WinSize表示实际画布的大小，VisibleSize表示可视区域的大小，所以，VisibleSize总是小于或者等于WinSize。值得注意的是，VisibleSize和VisibleOrigin是为Native服务的，仅在JSB上有效。另外，在CCBoot.js中实现了这样一行代码：

```
1  cc.winSize = cc.director.getWinSize();
```

因此，你可以通过cc.winSize的方式快速获取屏幕的大小。关于屏幕大小相关的内容，详见第7章。

3.1.5　执行游戏主循环

当游戏的底层环境设置好之后，游戏就可以以设定好的帧率（默认为60fps）进入主循环了。cc.Director定义了一个mainLoop方法，该方法控制着整个游戏的主循环。你可以在CCDirector.js中看到这样一段代码：

```
1 cc.Director = cc.Class.extend({
2     mainLoop: function () {
3         if (this._purgeDirectorInNextLoop) {
4             this._purgeDirectorInNextLoop = false;
5             this.purgeDirector();
6         }
7         else if (!this.invalid) {
8             this.drawScene();
9         }
10    }
11 });
```


首先，cc.Director会去检查下一帧是否需要清除自身，一般在调用cc.director.end()时被触发，这将清理和退出正在运行的场景，并且取消所有的调度器，删除所有的事件监听器，停止所有的动画以及清空缓存数据。

然后，开始绘制场景，Cocos2d-JS首先会去计算增量时间，即上一帧到现在所花的时间。理论上，它的值总是大于或等于游戏设定的帧率的倒数。所以，增量时间越大，游戏越显得卡顿。当增量时间大于一定值的时候，游戏开始出现明显的掉帧现象。

接着，cc.Director开始执行调度器中所被要求执行的任务。例如，《飞机大战》游戏中的飞机每隔0.5s发射子弹，若游戏以60fps来绘制，它大概是每隔30帧创建一颗子弹出来。若当前帧与上一次发射子弹刚好间隔了30帧，那这一次的子弹就会在当前帧被创建出来。

在执行调度器中的任务之后，cc.Director通过事件管理器触发cc.director.EVENT_AFTER_UPDATE事件，然后Cocos2d-JS开始检查是否有用户交互事件触发，例如触摸事件、鼠标实现、键盘事件、重力感应事件以及自定义事件，若有，则分发和处理这些事件。

事件分发完毕之后，渲染器把整个屏幕都清空掉，这是非常合理的。因为当渲染器清空完屏幕之后，导演开始判断是否有场景需要切换，若有，则切换到下一个场景，若没有，则遍历当前场景中的节点并更新节点的空间转换矩阵等信息，然后发送绘制指令给渲染器。接着，cc.Director触发cc.director.EVENT_AFTER_VISIT事件，并且会把所有的子节点按照层级从小到大进行排序，排完序之后，所有子节点的visit()方法将被触发。也正是在这个方法中，节点被绘制出来。

节点绘制完毕后，cc.Director触发cc.director.EVENT_AFTER_DRAW事件，最终更新当前帧率。

3.2 Cocos2d-JS 坐标系

渲染器渲染游戏对象时，需要获取游戏对象的位置。你应该知道，位置的概念都是相对的，它依赖于参照物，而在Cocos2d-JS中这个参照物就是坐标系。

在手机移动应用中，坐标系通常分为两种——屏幕坐标系和OpenGL坐标系。传统的原生应

用采用的是屏幕坐标，即以屏幕的左上角为原点，向右表示X轴的正方向，向下表示Y轴的正方向，这不管是在iOS、Android还是Windows Phone中，都是一样的。

然而，Cocos2d-JS并不是采用屏幕坐标系，而是选择了和OpenGL保持一致，即以屏幕的左下角为原点，向右表示X轴的正方向，向上表示Y轴的正方向。标准屏幕坐标系和Cocos2d-JS游戏引擎坐标系分别如图3-1和图3-2所示。

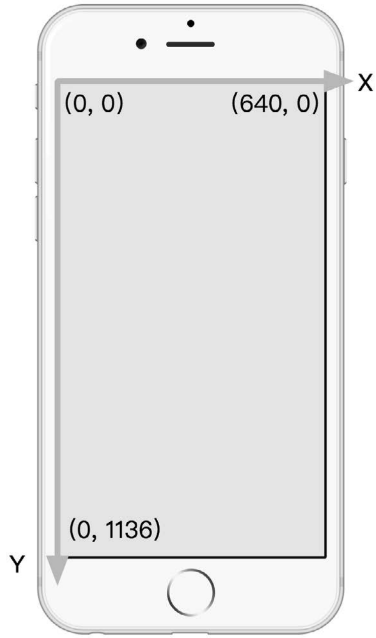


图3-1　标准屏幕坐标系

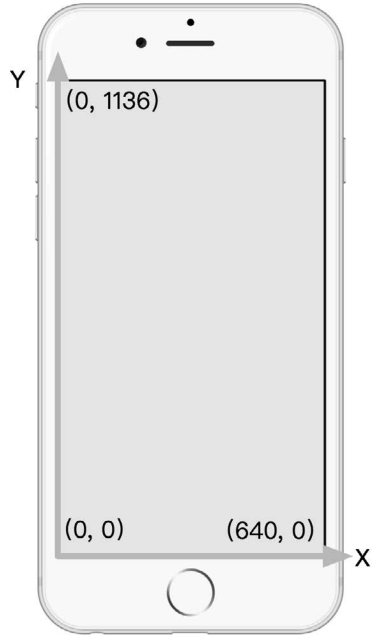


图3-2　Cocos引擎坐标系

另外，`cc.Director`提供了屏幕坐标和OpenGL坐标二者之间互相转换的方法，如下：

```
1  convertToGL = function (uiPoint)     // 转换[UI坐标为OpenGL/WebGL坐标]
2  convertToUI = function(glPoint)      // 转换[OpenGL/WebGL坐标为UI坐标]
```

除此之外，Cocos2d-JS中还有世界坐标系和本地坐标系的概念，详情可参见3.3节。

3.3　节点

你已经知道，在游戏主循环中，Cocos2d-JS会去遍历当前场景中的所有节点，那节点到底是什么东西呢？

节点，是一个代词，它是一个抽象概念类，在Cocos2d-JS中为`cc.Node`类。也就是说，所有`cc.Node`类及其子类，都可以归称为节点。`cc.Node`本身没有可视化表示形式，但凡是能被绘制，或者能包含其他节点的东西都是节点，例如下面马上要讲到的场景（`cc.Scene`）、层（`cc.Layer`）、

精灵（cc.Sprite）等。所有的节点都继承自cc.Node，cc.Node定义了所有节点共有的属性和方法。

3.3.1 基础属性

我将节点的位置、大小、锚点、层级、标签、名字归为基础属性，这是因为在日常开发中，它们是最常用的。

1. 位置

位置是节点最基础的属性，默认为cc.p(0, 0)，大部分的节点我们都需要给它设置一个位置。它的相关API如表3-1所示。

表3-1 位置相关的API

属　性	对应方法	参数或返回类型	说　明
x	setPositionX(posX)	Number	设置节点的X轴坐标
	getPositionX()	Number	获取节点的X轴坐标
y	setPositionY(posY)	Number	设置节点的Y轴坐标
	getPositionY()	Number	获取节点的Y轴坐标
_position	setPosition(newPosOrxValue, yValue)	cc.p(x, y)或Number	设置节点的坐标
	getPosition()	Number	获取节点的坐标

说明 属性前面带有下划线(_)的，均表示私有属性，子类不可访问，例如cc.Node的_position属性。

假设你现在需要将游戏中某个节点的位置设置为(100, 86)，并且获取出它的坐标，具体的实现代码如下：

```
1  node.x = 100;       // 设置[X轴坐标]
2  node.y = 86;        // 设置[Y轴坐标]
3  var posX = node.x;  // 获取[X轴坐标]
4  var posY = node.y;  // 获取[Y轴坐标]
```

上面的代码是通过直接操作属性的方式实现的，这也可以通过函数的方式来实现：

```
1  node.setPositionX(100);  // 设置[X轴坐标]
2  node.setPositionY(86);   // 设置[Y轴坐标]
3  node.getPositionX();     // 获取[X轴坐标]
4  node.getPositionY();     // 获取[Y轴坐标]
```

此外，你还可以通过下面的方式达到同样的目的：

```
1  node.setPosition(100, 86);          // 设置[坐标]
2  node.setPosition(cc.p(100, 86));    // 设置[坐标]
3  node.getPosition(); // 获取[坐标]，返回{cc.p}类型
```

你会发现，Cocos2d-JS提供了两种操作属性的方式，第一种是直接通过属性风格操作属性，第二种是通过函数调用。虽然通过属性操作的方式更加简便，但是属性风格的实现用到ECMAScript的getter和setter的特性，这两个特性在部分浏览器上面的效率要低于方法，所以在对性能要求比较高的地方还是建议使用方法进行访问。

说明　`cc.defineGetterSetter(proto,prop,getter,setter)`函数实现了ECMAScript的getter和setter特性，你可以在cocos2d-x/cocos/scripting/js-bindings/script/jsb_boot.js（Native实现）或cocos2d-x/web/cocos2d/core/platform/CCClass.js（Web实现）中看到它的实现。

---

2. 大小

在开发中，你可能需要设置或获取节点的大小，例如在做碰撞检测的时候。关于节点大小的相关API如表3-2所示。

表3-2　节点大小的相关API

属　　性	对应方法	参数或返回类型	说　　明
_contentSize	setContentSize(size, height)	Number	设置节点的内容大小
	getContentSize()	cc.size()数据结构	获取节点的内容大小

你还可以通过属性的方式快速访问节点的大小值，就像下面那样：

```
1  node.width = 100;           // 设置[节点的宽度]
2  node.height = 200;          // 设置[节点的高度]
3  var width = node.width;     // 获取[节点的宽度]
4  var height = node.height;   // 获取[节点的高度]
```

值得注意的是，这里设置的内容大小并不会改变节点的视觉效果，它只是一个属性，一般也被用到触摸事件中（见第5章）。

说明　cocos2d-x/web/cocos2d/core/base-nodes/BaseNodesPropertyDefine.js实现了节点的属性访问方式，例如`node.height`的实现如下：

```
1 var _p = cc.Node.prototype;
2 _p.height;
3 cc.defineGetterSetter(_p, "height", _p._getHeight, _p._setHeight);
```

除此之外，其他类的属性方式也是这样实现的，读者可自行阅读源码。

---

3. 锚点

现在你已经知道了节点有位置和大小属性，此时可以将某个节点想象为一张白纸，它的坐标就是`cc.p(100, 86)`，大小为`cc.size(125, 80)`。这个时候，我让你把这张白纸根据它的坐标将它钉在屏幕的某个位置，你会怎么做呢？我想你可能无从下手，因为你不确定把这张白纸的

哪个位置钉在屏幕上。

其实，这和白纸的锚点是有关系的。那么锚点是什么呢？锚点，它是一个百分比系数，其取值范围为[0, 0]到[1, 1]，边界包含0和1。每种类型的节点都会有自己的默认值，例如cc.Node类的默认节点为cc.p(0, 0)，表示节点的左下角定位在屏幕上你所设定的位置。所以，位置和锚点这两个属性共同决定了节点在屏幕中的摆放位置。通过下列代码可以设置和获取锚点：


```
1  node.setAnchorPoint(0.5, 0.5);          // 设置[锚点]
2  var anchor = node.getAnchorPoint();     // 获取[锚点]，返回{cc.p}类型
```

上面的第1行代码将白纸的锚点设置为(0.5, 0.5)，这可以非常清楚地得出将白纸的中心点钉在屏幕上，但是，倘若我们将白纸的锚点设置为(0.3, 0.2)，那么纸张定位在屏幕上的定位点（称为原点）应该这么算：

X轴坐标：节点宽度 × 锚点x值

Y轴坐标：节点高度 × 锚点y值

因此，得出此时白纸的原点是cc.p(125 * 0.3, 80 * 0.2)。当然，你也可以通过cc.Node的getAnchorPointInPoints()函数直接获取原点。

除此之外，锚点还决定节点变换操作，例如旋转和缩放等。因为动作（见第4章）是基于变换操作的，所以锚点也是动作运转的围绕点。图3-3和图3-4中的节点都被旋转了–30°，不同的是它们的锚点分别是cc.p(0.5, 0.5)和cc.p(0, 0)。



图3-3 锚点为cc.p(0.5,0.5)，旋转–30°

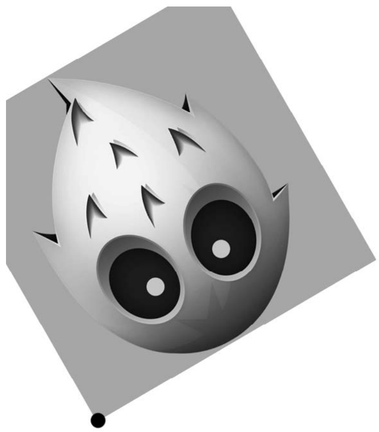


图3-4 锚点为cc.p(0,0)，旋转–30°

4. 层级

若当前游戏中只有一个节点，渲染器可以直接对此节点进行绘制。但是，倘若游戏中有两个以上的节点需要绘制，那么渲染器可能会不知所措了，因为它并不知道要将哪个节点绘制在屏幕的最上方，哪个节点绘制在屏幕的最下方。所以，每个节点都会有一个层级属性，用来告诉渲染

器节点的绘制顺序。

层级又分为全局层级和本地层级两种，全局层级用globalZOrder表示，本地层级用localZOrder表示，全局层级的绘制优先级高于本地层级。获取和设置层级的相关API如表3-3所示。

表3-3　获取和设置层级的相关API

属　　性	对应方法	参数或返回类型	说　　明
zIndex	setLocalZOrder(localZOrder)	Number	设置节点的本地层级
	getLocalZOrder()	Number	获取节点的本地层级
_globalZOrder	setGlobalZOrder(globalZOrder)	Number	设置节点的全局层级
	getGlobalZOrder()	Number	获取节点的全局层级

一般情况下，层级保持默认值即可。在开发中，能少用层级的就尽量少用层级，免得节点层级关系乱七八糟的，给自己带来麻烦。若在一些非用层级不可的地方，那么建议定义一个类似枚举的层级对象，像下面这样：

```
1  var NodeZorder = NodeZorder || {};
2  NodeZorder.HERO = 100;      // 角色对象层级
3  NodeZorder.ROLE = 200;      // 角色对象层级
4  // ......
5  this.addChild(node, NodeZorder.HERO);   // 将节点添加到当前层中
```

5. 标签

在游戏中可能同时存在好多个节点，你可以为每个节点都贴上一个标签，方便区分它们。cc.Node提供了一个叫tag的属性，默认值为–1，是Number类型。你可以通过如表3-4所示的API设置和获取节点的tag值。

表3-4　设置和获取节点tag值的API

属　　性	对应方法	参数或返回类型	说　　明
tag	setTag(tag)	Number	设置节点的标签
	getTag()	Number	获取节点的标签

注意　cc.Node的tag值必须为整型，否则在JSB上会报错，这是因为Cocos2d-x API设计的就是整型的。

6. 名字

或许你已经意识到了，用tag标记节点的方式可能不能完全满足开发需求。因为tag为整型类型，虽然我们可以通过定义常量的方式将tag值中转一下，但是若tag值相等的话，节点依旧无法根据tag值区分，就像下面这样：

```
1  MAN = 1;
2  WOMAN = 1;
3  var man = new cc.Node();      // 创建[节点]
4  man.setTag(MAN);              // 设置[tag属性]
5  var woMan = new cc.Node();    // 创建[节点]
6  woMan.setTag(WOMAN);          // 设置[tag属性]
```

所以，Cocos2d-JS也为节点设计了name属性，它是字符串类型的。你可以通过表3-5所示的API设置和获取它。

表3-5 name属性对应的方法

属　性	对应方法	参数或返回类型	说　明
name	setName(name) getName()	String String	设置节点的名字 获取节点的名字

3.3.2 图形属性

除了基础属性之外，Cocos2d-JS还为cc.Node提供了一些和图形相关的属性，这些属性直接影响每个节点的图形效果，下面我尝试着将这些和图形相关的属性提取出来并概要介绍一下它们。

1. 旋转

引擎为节点设计了rotation属性来控制节点的旋转角度，其默认值为0，它是Number类型的。另外，值得说明的是，在Cocos2d-JS的坐标系中，垂直X轴向上表示0°，垂直Y轴向右为90°，rotation的值为正数表示顺时针旋转，负数表示逆时针旋转。表3-6列出了旋转相关的属性。

表3-6 旋转相关的属性

属　性	对应方法	参数或返回类型	说　明
rotation	setRotation(angle) getRotation()	Number Number	设置节点的旋转角度 获取节点的旋转角度
rotationX	setRotationX(angle) getRotationX()	Number Number	设置节点X轴的旋转角度 获取节点X轴的旋转角度
rotationY	setRotationY(angle) getRotationY()	Number Number	设置节点Y轴的旋转角度 获取节点Y轴的旋转角度

2. 缩放

和旋转类似，你可以通过节点的scale属性来控制节点渲染时的缩放比例，即视觉上节点的大小。它的取值范围是R（全体实数），默认值为1。设置为负数的话，引擎将会反向渲染显示。例如要水平翻转，只需要调用setScaleX(-1)即可。表3-7列出了缩放相关的属性。

表3-7　缩放相关的属性

属　　性	对应方法	参数或返回类型	说　　明
scale	setScale(radio)	Number	设置节点的缩放倍数
	getScale()	Number	获取节点的缩放倍数
scaleX	setScaleX(radio)	Number	设置节点X轴的缩放倍数
	getScaleX()	Number	获取节点X轴的缩放倍数
scaleY	setScaleY(radio)	Number	设置节点Y轴的缩放倍数
	getScaleY()	Number	获取节点Y轴的缩放倍数

3. 倾斜

除了旋转和缩放之外，引擎还提供skew属性来控制节点的倾斜度，对应的API如表3-8所示。

表3-8　**skew**属性对应的API

属　　性	对应方法	参数或返回类型	说　　明
skewX	setSkewX(skew)	Number	设置节点的X轴倾斜度
	getSkewX()	Number	获取节点的X轴倾斜度
skewY	setSkewY(skew)	Number	设置节点的Y轴倾斜度
	getSkewY()	Number	获取节点的Y轴倾斜度

4. 可见性

有的时候，你可能需要将节点隐藏起来，使其不可见。节点的visible属性就是用来控制节点对象是否可见的，当visible值为false时，引擎将不会对该节点进行渲染。visible属性对应的API如表3-9所示。

表3-9　**visible**属性对应的API

属　　性	对应方法	参数或返回类型	说　　明
visible	setVisible(isVisible)	Boolean	设置节点是否可见
	isVisible()	Boolean	获取节点是否可见

5. 不透明

在游戏内部需要实现一些半透明效果，节点的opacity属性就是为此设计的。它的取值区间是[0, 255]，0表示全透明，255表示完全不透明。另外，cascadeOpacity属性可以控制opacity是否进行子节点关联，使opacity值也作用于其子节点上。opacity属性对应的API如表3-10所示。

表3-10　**opacity**属性对应的API

属　　性	对应方法	参数或返回类型	说　　明
opacity	setOpacity(opacity)	Number	设置不透明度
	getOpacity(opacity)	Number	获取不透明度
cascadeOpacity	setCascadeOpacity(isCascadeOpacity)	Boolean	设置是否级联不透明度
	isCascadeOpacity()	Boolean	获取是否级联不透明度

注意 和设置可见性不同，设置节点透明度为0后，节点虽然看不到了，但这只是进行透明度混合运算的结果，引擎还是会渲染该节点。

6. 颜色

节点的`color`属性可以设置节点显示的颜色，参数为`cc.color`对象。`cc.color`接收一个RGB参数，表示红（R）、绿（G）、蓝（B）三个颜色通道，例如`cc.color(255,100,200)`，每个通道的取值区间是[0, 255]。

`cascadeColor`属性表示是否进行子节点关联，即是否使当前节点的颜色值也作用到其子节点上。`opacityModifyRGB`属性用于指定颜色值是否要跟着不透明度进行改变。颜色相关的属性对应的API如表3-11所示。

表3-11 颜色相关的属性对应的API

属　性	对应方法	参数或返回类型	说　明
color	setColor(color)	cc.color	设置节点的颜色
	getColor()	cc.color	获取节点的颜色
cascadeColor	setCascadeColor(b)	Boolean	设置是否级联颜色值
	isCascadeColor()	Boolean	获取是否级联颜色值
opacityModifyRGB	setOpacityModifyRGB(b)	Boolean	设置颜色是否跟随不透明度进行改变
	isOpacityModifyRGB()	Boolean	获取颜色是否跟随不透明度进行改变

7. 渲染程序

在游戏里面有的时候需要用到一些渲染特效，这些特效通常是一些shader（着色器）程序，节点的`shaderProgram`属性就是因此而生的，相关的API如表3-12所示。

表3-12 `shaderProgram`属性相关的API

属　性	对应方法	参数或返回类型	说　明
shaderProgram	setShaderProgram(sp)	cc.GLProgram	设置节点的着色器程序
	getShaderProgram()	cc.GLProgram	获取节点的着色器程序

3.3.3 其他属性

除了以上属性之外，Cocos2d-JS还为`cc.Node`提供了其他属性，比如父/子节点相关的属性、`running`属性、`actionManager`属性、`arrivalOrder`属性等。

1. 父/子节点相关

引擎中层（Layer）通常包含其他的节点，同时层又添加到场景（Scene）里面，这样它们便有了父子关系，场景是层的父节点，其他添加到层里面的节点就是层的子节点。节点的`parent`

属性表示的就是该节点的父节点对象，而children属性表示的是当前节点的所有子节点的一个集合，childrenCount属性表示当前节点拥有的子节点数量。相关的API如表3-13所示。

表3-13　父/子节点相关的API

属　　性	对应方法	参数或返回类型	说　　明
parent	setParent(parentNode)	cc.Node	设置父节点对象
	getParent()	cc.Node	获取父节点对象
children	getChildren()	Array	获取所有子节点对象集合
childrenCount	getChildrenCount()	Number	获取子节点数量

注意　children属性和childrenCount属性是只读属性，节点的子节点只能通过addChild()方法添加，在添加的同时childrenCount属性会自动计数。

2. 运行状态

节点的running属性用于判断该节点是否处于运行状态。当节点的生命周期进入onEnter时，running的值为true；当节点的生命周期进入onExit时，running的值为false。running属性对应的API如表3-14所示。

表3-14　**running**属性对应的API

属　　性	对应方法	参数或返回类型	说　　明
running	isRunning()	Boolean	判断节点是否处于运行状态

3. 动作管理器

每个节点都会有一个动作管理器来管理节点的动作，actionManager属性表示的就是当前节点的动作管理器对象，对应的API如表3-15所示。

表3-15　**actionManager**属性对应的API

属　　性	对应方法	参数或返回类型	说　　明
actionManager	setActionManager(am)	cc.ActionManager	设置节点的动作管理器
	getActionManager()	cc.ActionManager	获取节点的动作管理器

4. 到达顺序

引擎在绘制节点对象的时候会根据层级进行排序，但是有的节点的层级是相同的，而arrivalOrder属性就是用来对层级相同的节点进行排序的。arrivalOrder值越小的节点将被引擎优先渲染。该属性属于引擎内部使用，不建议修改，对应的API如表3-16所示。

表3-16 **arrivalOrder**属性对应的API

属 性	对应方法	参数或返回类型	说 明
arrivalOrder	setOrderOfArrival(order)	Number	设置节点的到达顺序
	getOrderOfArrival()	Number	获取节点的到达顺序

3.3.4 常用函数

介绍完cc.Node的属性之后，一起来看看cc.Node的常用函数，这里我依旧将它们按功能进行了分类。

1. 属性配置

cc.Node提供了attr()函数供开发者快速配置属性，像下面那样使用：

```
1  node.attr({
2      x : 200,          // 设置[节点的X轴坐标]
3      y : 300,          // 设置[节点的Y轴坐标]
4      scale : 1.5,      // 设置[节点的缩放比例]
5      rotation : 45     // 设置[节点的旋转角度]
6  });
```

2. 节点操作

操作节点的大概有“增、删、改、查”4个大功能，下面简要介绍这4个功能。

- 增

当创建好一个节点之后，你可以把它添加到父节点中，添加子节点的API如下：

```
1  addChild(child, localZOrder, tag)
```

child为子节点是毫无疑问的，localZOrder和tag表示本地层级和标签，这在前面已经说过了。不过，还需要说明的是，localZOrder的默认值为0，tag的默认值为–1，并且它们都可以省略。

- 删

cc.Node提供了4个删除节点的相关API，具体如下：

```
1  parent.removeChild(child, cleanup);       // 删除[子节点]
2  parent.removeChildByTag(tag, cleanup);    // 根据tag删除[子节点]
3  parent.removeAllChildren(cleanup);        // 删除[所有子节点]
4  node.removeFromParent(cleanup);           // 子节点自己从父节点删除
```

这4个API的cleanup参数都是布尔值，表示是否清除节点中所有的动作以及回调函数，默认值为true，也可省略。

- 改

这里的改，实际上是重新对节点的arrivalOrder和localZOrder进行赋值。与node.setLocalZOrder(localZOrder)函数不同的是，它对arrivalOrder属性也重新赋值了。所

以，理论上若你想对节点赋予一个新的渲染顺序，通过下面的方法会更好一些。当然，你得保证父节点parent中有child这个子节点：

```
1  parent.reorderChild(child, zOrder)
```

- 查

记得在前面提到过，cc.Node为节点设置了tag和name属性，通过这两个属性，便可以在父节点中找到想要的子节点，相关API如下：

```
1  node.getChildByTag(tag)
2  node.getChildByName(name)
```

除此之外，cc.Node还提供了返回子节点数组以及子节点个数的API，如下：

```
1  node.getChildren();          // 获取所有子节点，返回一个数组
2  node.getChildrenCount();     // 获取子节点个数
```

3. 暂停和恢复

通过cc.Node的pause()和resume()函数，可以暂停和恢复节点的动作以及调度器，这一般用在暂停游戏和恢复游戏的功能上。你可以把需要暂停和恢复的节点放在一个数组中，然后遍历这个数组，再调用数组中节点的pause()或resume()函数即可，相关API如下：

```
1  node.pause();    // [暂停]节点的动作以及调度器
2  node.resume();  // [恢复]节点的动作以及调度器
```

4. 动作操作

每一个节点都可以运行动作（参见第4章），cc.Node提供了如下和动作相关的API：

```
1  node.runAction(action);      // 运行指定动作
2  node.stopAction(action);     // 停止指定动作
3  node.stopAllActions();       // 停止所有动作
4  node.stopActionByTag(99);    // 根据动作的tag值停止动作
5  var action = node.getActionByTag(99);    // 根据tag值在动作列表中取出动作
```

5. 内存管理

内存管理，一直是程序员口中的热门话题。在C++中，动态地分配内存可谓是一把双刃剑，直接访问内存地址的方式提高了程序的性能，但是比较遗憾的是，每次手动分配和释放内存，都可能带来没有正确分配和释放内存所造成的一系列问题，例如野指针、重复释放、内存泄漏等重大让人头疼的问题。

人们尝试着用不同的解决方案去避免这个问题，比较常见的有智能指针、自动垃圾回收等，而Cocos游戏引擎采用的内存管理方式就是自动垃圾回收的方式。注意，这里是自动回收，这意味着你并不需要自己手动管理对象的内存。

Cocos引擎采用的内存管理机制实际上来源于Mac OS X 10.7与iOS 5中引入的一项新技术——ARC，中文译为自动引用计数，是Automatic Reference Counting的缩写。ARC在Cocos2d-x中的原理大概是这样的：

“在Cocos2d-x中，每个对象都有一个_referenceCount属性，该对象被引用时，_referenceCount会执行+1操作，例如调用node.addChild(child)函数。当对象被释放时，_referenceCount会执行−1操作，例如调用node.removeChild(child)函数。在每一帧绘制完成后，对象回收池（AutoreleasePool）会去检查回收池中所有对象的_referenceCount值，当对象的_referenceCount值为0时，对象回收池便会回收释放的内存。”

虽然Cocos引擎提供了自动内存回收的机制，但是你仍然可以通过如下API管理内存。当然，它们仅在JSB中有效，并且在使用它们的时候，你必须明白自己在做什么，否则会照成内存泄露等严重问题：

```
1  node.retain();       // 使节点的引用计数（_referenceCount）加1
2  node.release();      // 使节点的引用计数（_referenceCount）减1
```

6. 调度器

在开发中，你可能需要游戏每隔一段时间做某一件事情，比如每隔0.1s发射一颗子弹，这样的需求就需要调度器来实现。cc.Node为调度器提供了相关的API，你可以通过如下代码开启默认调度器：

```
1  node.scheduleUpdate();
2  node.update = function(dt){
3      cc.log("每一帧都会执行，每一帧时间间隔为：",dt);
4  }
```

需要注意的是，当一个节点调用了scheduleUpdate()函数之后，你必须重写此节点的update(dt)函数。在update(dt)中，dt参数表示时间间隔，一般为每一帧的时间。你可以用它做一个计时器，在帧率为60fps的情况下，执行了60次的update函数，就是1秒了。

此外，你还可以通过schedule函数启动一个自定义调度器，其API如下：

```
1  node.schedule(callback, interval, repeat, delay, key)
```

其中，callback为调度器的回调函数，是function类型。interval、repeat和delay分别表示时间间隔、重复次数、延时多少秒开始执行，它们都是Number类型。而key为String类型，是调度器的唯一标识符。除了callback外，其余参数都是可省略的，这说明它们都有一个默认值。interval的默认值为0，若省略它，应该考虑使用默认的update(dt)调度器。repeat的默认值为cc.REPEAT_FOREVER，是一个正无穷的数，表示无限循环。delay的默认值也为0，下面举个使用schedule函数启动的调度器例子：

```
1  callback = function(){
2      cc.log("自定义定时器...每隔0.2s执行一次...");
3  };
4  node.schedule(callback, 0.2);
```

cc.Node还提供了只调度一次的调度器, 其参数和用法都与schedule函数一致, 其API如下:

```
1  node.scheduleOnce(callback, delay, key)
```

除此之外，cc.Node拥有两个卸载调度器的函数，一个是卸载指定调度器，一个是卸载所有

调度器，对应API如下：

```
1  node.unschedule(callback);  // 卸载callback调度器，callback即调度器回调函数
2  node.unscheduleAllCallbacks();  // 卸载node节点所有的调度器
```

7. 坐标转换

前面提到过，Cocos2d-JS中有本地坐标和世界坐标的概念，本地坐标指的是以父节点的左下角为原点的坐标参考系，而世界坐标以标准的Cocos2d-JS坐标系作为参考系。例如，当前游戏中有Node_A和Node_B两个精灵节点（因为精灵节点的默认锚点是`cc.p(0.5,0.5)`，所以精灵节点的原点在其自身的正中心，见3.6节），它们的大小都是`cc.size(200,200)`，Node_B是Node_A的子节点。Node_A的坐标为`cc.p(200,200)`，Node_B的坐标为`cc.p(100,100)`，那么它们在游戏中的坐标位置如图3-5所示。

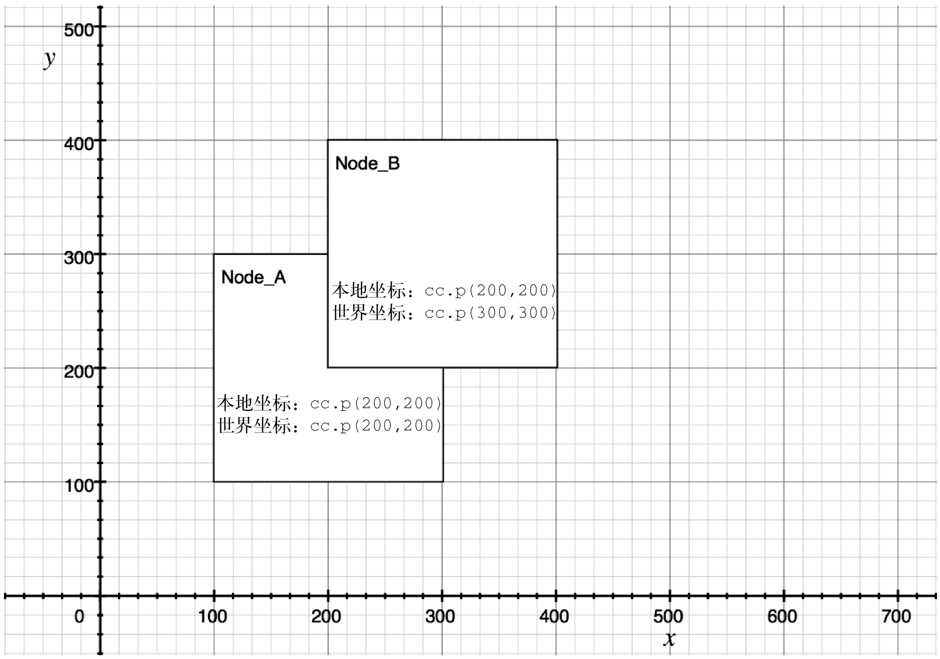


图3-5　本地坐标和世界坐标

`cc.Node`提供了本地坐标和世界坐标转换的相关代码，具体如下：

```
1  // 本标坐标转为世界坐标
2  var toWorldPos = nodeParent.convertToWorldSpace(node.getPosition());
3  // 本地坐标转为世界坐标（受锚点影响）
4  var toWorldPosAR = nodeParent.convertToWorldSpaceAR(node.getPosition());
5  // 世界坐标转为本地坐标
6  var toNodePos = node_A.convertToNodeSpace(node_B);
7  // 世界坐标转为本地坐标（受锚点影响）
8  var toNodePosAR = node_A.convertToNodeSpaceAR(node_B);
```

8. 边框区域

cc.Node提供了两个和边框区域有关的API，具体如下：

```
1  node.getBoundingBox();          // 返回节点的本地坐标系的外边框
2  node.getBoundingBoxToWorld();   // 返回节点的世界坐标系的边框
```

它们的返回值都是一个cc.rect(x,y,width,height)的数据结构，其中x、y分别表示节点的X轴和Y轴的坐标，width和height表示节点的宽度和高度。

3.4　场景

我总是喜欢把游戏和电影联想起来，因为游戏的开发和电影的制作是那么相似。当然，我更喜欢把Cocos2d-JS游戏和京剧联想起来，因为你会发现，Cocos2d-JS游戏的组织结构和京剧几乎一样，就比如Cocos2d-JS中的cc.Scene，它就对应着京剧中的舞台，供演员们表演。舞台一次展示一个，场景也只能一次运行一个，所以，在Cocos2d-JS游戏中，至少存在一个场景供游戏运作。先来看下cc.Scene的构造函数，具体如下：

```
1  cc.Scene = cc.Node.extend({
2      _className:"Scene",
3      ctor:function () {
4          cc.Node.prototype.ctor.call(this);                // 调用[父类构造函数]
5          this._ignoreAnchorPointForPosition = true;        // 设置[忽略锚点]
6          this.setAnchorPoint(0.5, 0.5);                    // 设置[锚点]
7          this.setContentSize(cc.director.getWinSize()); // 设置[大小]
8      }
9  });
```

从上面的代码不难发现，场景（cc.Scene）类十分简单，它只是作为层（cc.Layer）（见3.5节）的容器。从第5行代码可以看到，场景有忽略锚点属性，默认值为true。虽然在第6行代码中，锚点被设置为(0.5, 0.5)，但是这已经没什么用了，因为锚点已经被忽略了。通常情况下，你不应该去更改场景的锚点和忽略锚点的值，它只是提供给引擎自己使用的。

在实际开发中，你应该学会根据功能设计场景。例如，游戏主页面、关卡选择页面、游戏玩法页面、设置页面、游戏结束页面等都设计成一个场景，页面的切换实际上就是场景的切换。

3.4.1　再探导演——场景管理

作为层的容器，场景可以添加层，但是不能再添加场景。场景与场景的切换，只能通过cc.Director来实现。使用cc.Director管理场景的相关方法如下：

```
1  cc.Director = cc.Class.extend({
2      getRunningScene : function();          // 获取[正在运行的场景]
3      runScene : function(scene);            // 运行[指定场景]
4      pushScene : function(scene);           // 压入[指定场景]
5      popScene : function();                 // 弹出[顶部场景]
6      popToRootScene : function();           // 弹出[到根场景]
```

```
7       popToSceneStackLevel : function(level); // 弹出[到指定栈级场景]
8   );
```

cc.Director提供了一个getRunningScene()方法，这让你在任何地方都可以获取到当前正在运行的场景。

再者，你会发现，cc.Director只提供了一种方法从一个场景切换到另一个场景，这和Cocos2d-x显得有些出入了。Cocos2d-x首次运行场景用runWithScene()，场景和场景之间的切换通过replaceScene()方法替换，而Cocos2d-JS废弃了runWithScene()和replaceScene()方法，简化了场景的管理，所有场景之间的切换都采用runScene()方法。

另外，cc.Director还提供场景栈来管理场景，如图3-6所示。

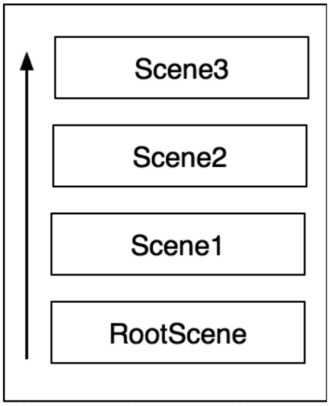


图3-6　场景栈

通常情况下，pushScene用来压入一些占用内存比较小的场景。这是因为pushScene会终止当前正在运行的场景，并且把它放入到场景栈当中去，将它“冷藏”起来，然后才运行新的场景，当前的场景并没有被释放掉。因此，一般应该选择压入游戏设置等这一类占用内存较小的场景，但如果你压入的场景所使用的内存过大，则可能引起设备内存不足，Cocos2d-JS发现内存不足，则会释放被“冷藏”起来的场景中所用到的资源所占的内存，这将造成不可控制的危害。

pop相关操作场景的方法大同小异，都是将经过pushScene的场景从堆栈（内存）中pop出来执行，但前提是堆栈内存中存在此场景，而当前执行的场景将被删除。

3.4.2　切换特效

场景之间的切换，Cocos2d-JS提供了大几十种的过渡效果，例如四方推入、淡入淡出等，示例如下：

```
1  // 四方推入
2  cc.director.runScene(new cc.TransitionMoveInL(time, scene));    // [从左往右]推入
3  cc.director.runScene(new cc.TransitionMoveInR(time, scene));    // [从右往左]推入
```

```
4 cc.director.runScene(new cc.TransitionMoveInT(time, scene));     // [从下往上]推入
5 cc.director.runScene(new cc.TransitionMoveInB(time, scene));     // [从上往下]推入
6 // 淡入淡出
7 cc.director.runScene(new cc.TransitionFade(time,scene));
```

其中，参数time是过渡时长，scene为目标场景。关于更多的切换特效，本书不一一列举，读者可参考js-tests中的TransitionsTest示例。

3.4.3 场景的生命周期

在Cocos2d-JS引擎中，游戏场景在运行的过程中会经历几个不同的阶段，这些阶段就是我们所说的生命周期。每一个阶段皆表示场景的一个状态，每个状态都有不同的表现。场景从构造到最终显示在屏幕上，再到场景销毁以及退出经历了以下几个阶段：

```
1  var MainMenuScene = cc.Scene.extend({
2      ctor : function(){            // 构造函数
3          // TODO something
4      },
5      onEnter : function () {       // 场景进入
6          this._super();
7          // TODO something
8      },
9      onEnterTransitionDidFinish: function () {    // 场景进入过渡效果执行完毕
10         this._super();
11         // TODO something
12     },
13     onExitTransitionDidStart: function(){        // 场景退出过渡效果开始执行
14         // TODO something
15         this._super();
16     },
17     onExit : function(){          // 场景退出
18         // TODO something
19         this._super();
20     }
21 });
```

其中，this._super()函数为调用父类的当前方法。值得注意的是，在退出相关的生命周期函数中，应该将this._super()函数放在最后调用，遵循“先进后出”的原则。

当两个场景进行切换的时候，整个过程的生命周期变化顺序如下（其中SceneA为当前场景，SceneB为即将切换过去的场景）：

```
1 SceneA : ctor
2 SceneA : onEnter
3 SceneA : onEnterTransitionDidFinish
4 SceneB : ctor
5 SceneA : onExitTransitionDidStart
6 SceneA : onExit
7 SceneB : onEnter
8 SceneB : onEnterTransitionDidFinish
```

可以看到，Cocos2d-JS会先调用SceneB的ctor构造函数，然后继续调用SceneA的onExitTransitionDidStart和onExit函数，最后在SceneB的onEnter函数中将SceneB渲染出来。不难发现，SceneB的ctor构造函数是两个场景切换的关键所在，它是两个场景切换时的临界点，所以，你不应该在场景的ctor函数中做一些对内存开销大的操作（例如加载资源等），这会导致两个场景的资源都被加载到内存，从而可能导致内存不足，引起程序崩溃。

3.5　层

层是一个容器，它最大的作用就是管理场景中的节点。层一般作为场景的子节点，当然，它也可以作为其他节点的子节点，进一步管理子节点的子节点，但这并不多见。下面为cc.Layer的构造函数：

```
1  cc.Layer = cc.Node.extend({
2      _className: "Layer",
3      ctor: function () {
4          var nodep = cc.Node.prototype;                     // 获取[父类原型]
5          nodep.ctor.call(this);                             // 调用[父类构造函数]
6          this._ignoreAnchorPointForPosition = true;         // 设置[忽略锚点]
7          nodep.setAnchorPoint.call(this, 0.5, 0.5);         // 设置[锚点]
8          nodep.setContentSize.call(this, cc.winSize);       // 设置[大小]
9      }
10 }
```

从上述代码的第6行到第8行可以得到，层的一些默认设置和场景是一样的，也是默认忽略锚点，锚点为(0.5, 0.5)，且大小为当前屏幕的大小。

实际上，Cocos2d-JS提供了4种层，cc.Layer只是其中最基本的一种。除此之外，其他3种层分别是cc.LayerColor、cc.LayerGradient和cc.LayerMultiplex。值得注意的是，它们都继承cc.Layer。

1. cc.LayerColor

cc.LayerColor作为一个可以携带颜色的层，在平常开发中非常常用，例如游戏中的Dialog模态对话框中背景半透明变暗的实现。通常，我们会创建一个cc.LayerColor，指定颜色为cc.color(0,0,0,138)，大小为当前屏幕的大小，就像下面这样：

```
1  var width = cc.winSize.width;        // 获取[屏幕宽度]
2  var height = cc.winSize.height;      // 获取[屏幕高度]
3  var dialog = new cc.LayerColor(cc.color(0, 0, 0, 138), width, height);
4  this.addChild(dialog);
```

实际上，上述第3行代码传递进去的width和height是可省略的，这可以从cc.LayerColor中的init方法找到答案：

```
1  cc.LayerColor = cc.Layer.extend({
2      _blendFunc: null,
3      _className: "LayerColor",
```

```
4       //......
5       init: function (color, width, height) {
9           var winSize = cc.director.getWinSize();
10          color = color || cc.color(0, 0, 0, 255);
11          width = width === undefined ? winSize.width : width;
12          height = height === undefined ? winSize.height : height;
13
14          var locRealColor = this._realColor;
15          locRealColor.r = color.r;
16          locRealColor.g = color.g;
17          locRealColor.b = color.b;
18          this._realOpacity = color.a;
19          this._renderCmd.setDirtyFlag(cc.Node._dirtyFlags.colorDirty
            cc.Node._dirtyFlags.opacityDirty);
20
21          cc.LayerColor.prototype.setContentSize.call(this, width, height);
22          return true;
23      }
24      //......
25 });
```

在上面代码的第10行中可以看到，若你在创建cc.LayerColor的时候没有指定颜色，那么cc.LayerColor的颜色将被强制设置为cc.color(0,0,0,255)，opacity被设置为255。

在第11行和第12行中，Cocos2d-JS会去判断你是否给cc.LayerColor指定了宽度和高度，若没有，则强制设置为屏幕的大小。所以，刚刚创建的cc.LayerColor就可以简化成下面这样：

```
1  var dialog = new cc.LayerColor(cc.color(0, 0, 0, 138));
2  this.addChild(dialog);
```

2. cc.LayerGradient

cc.LayerGradient为一个颜色渐变层，你可以像下面这样创建一个从红色渐变到绿色的层：

```
1  var start = cc.color.RED;
2  var end = cc.color.GREEN;
3  var vector = cc.p(-1, 1);
4  var layer = new cc.LayerGradient(start, end, vector);
5  scene.addChild(layer);
```

上面第1行和第2行分别定义了两个颜色对象，作为开始和结束的颜色。第3行定义了一个矢量，用于表示颜色过渡的方向。

3. cc.LayerMultiplex

cc.LayerMultiplex对象可以用来管理其他的cc.Layer对象。假设你的游戏场景里面有很多层，这些层有的时候需要切换显示，这时候cc.LayerMultiplex就可以派上用场了，你只需将要切换的层放在cc.LayerMultiplex对象中，然后就可以通过switchTo(n)函数来切换到不同的层上，switchTo(n)函数的参数n表示要切换的cc.Layer的索引。例如：

```
1  var layer1 = new cc.LayerGradient(cc.color.RED, cc.color.GREEN, cc.p(-1, 1));
2  var layer2 = new cc.LayerColor(cc.color.RED);
3  var layer3 = new cc.LayerColor(new cc.Color(0,255,0,255));
```

```
4  var layerMultiplex = new cc.LayerMultiplex([layer1, layer2, layer3]);
5  layerMultiplex.switchTo(1);
6  scene.addChild(layerMultiplex);
```

上面第1行到第3行分别定义了3个不同的cc.Layer对象，第4行创建了一个cc.LayerMultiplex对象，cc.LayerMultiplex构造函数接收一个数组作为参数，将前面创建的三个层添加到cc.LayerMultiplex上。第5行切换到你要显示的层，其参数是构造时传递进去的数组的索引。这里需要注意，索引不可越界。

3.6　精灵

精灵是游戏引擎中一个不可或缺的元素，你可以用它来表示游戏中的背景、主角、血条等对象。每个精灵一般都关联着一张纹理贴图，也就是Texture2D对象，所以，在cc.Sprite类中，有一个texture属性。cc.Sprite渲染的矩形区域的内容可能是来自这张纹理贴图的某块区域或者是全部，当然，也可以是一个与纹理无关的纯色区域。Cocos2d-JS提供了4种创建精灵的方式，具体如下。

3.6.1　通过图片资源创建

在创建精灵的时候，你可以直接将要显示的图片路径当成参数传递给构造函数，就像下面这样：

```
1  var node = new cc.Sprite("res/node_256.png");
```

或许在有些地方，你还会看到使用如下方式创建精灵：

```
1  var node = new cc.Sprite(res.sh_node_256_png);
```

实际上，上面两行代码的效果都一样，其中res对象只是对资源做了一个定义，方便管理资源。除此之外，它们并没有什么区别。但实际上，我不推荐你用单张图片的方式创建精灵，因为这会浪费一定的内存，详见第11章。

3.6.2　通过纹理创建

与通过图片直接创建精灵不同的是，通过纹理创建精灵需要你将要使用的图片添加到纹理缓存里面去，然后根据返回的纹理对象创建精灵，具体如下：

```
1  var textrue = cc.textureCache.addImage(res.sh_node_256_png);
2  var node = new cc.Sprite(textrue);
```

不过，需要说明的是，通过上面的第1行代码，res.sh_node_256_png所对应的图片资源已经被加载到内存中了，所以，此时再通过如下代码创建精灵的时候就会快一些：

```
1  var node = new cc.Sprite(res.sh_node_256_png);
```

3.6.3 通过精灵表单创建

精灵表单是一个plist配置文件和一张图片资源文件，它将一系列图片资源合成一张大图，然后将大小、坐标等配置保存在plist中。它是为了节省纹理所占用的内存而设计的，所以，通过精灵表单的方式是最合理、最好的方式，除非你对内存的开销不是很在乎。精灵和精灵表单的关系如图3-7所示。

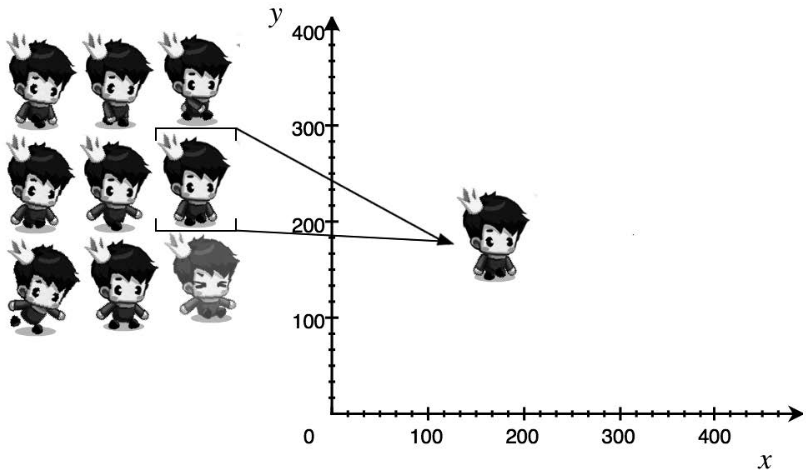


图3-7 精灵表单和精灵的关系

使用精灵表单创建精灵，需要先将plist和图片文件加载到内存，然后再创建，相关代码如下：

```
1  cc.spriteFrameCache.addSpriteFrames(res.u11_prince_plist, res.u11_prince_png);
2  var node = new cc.Sprite("#prince_stand_1.png");
```

说明 关于精灵表单的进一步说明，请详见第11章。

3.6.4 创建带有颜色的精灵

前面介绍了几种和图片资源相关联的精灵，但是在一些情况下，你可能只是需要一个色块这样的精灵。那么，Cocos2d-JS也提供了实现方式，相关代码如下：

```
1  var node = new cc.Sprite();
2  node.setColor(cc.color(255, 200, 10));          // 设置颜色
3  node.setTextureRect(cc.rect(0, 0, 200, 200));   // 设置纹理矩形
```

需要注意的是，当你想创建一个纯色块精灵时，你应该像第3行代码那样，设置它的纹理矩形，而不是调用`setContentSize(cc.size(width,height))`函数。

3.7　场景树

你已经知道，Cocos2d-JS游戏中有场景的概念，场景下面组织着各种层，层里面又放着精灵、标签（Label）、UI等节点。非常显然，这是树的结构。树是数据结构中非常重要的一个概念，它通常被用来组织一些具有层次结构的数据，而Cocos2d-JS正是结合了树的概念，实现了对节点的查找、遍历、修改和排序。Cocos游戏场景树如图3-8所示。

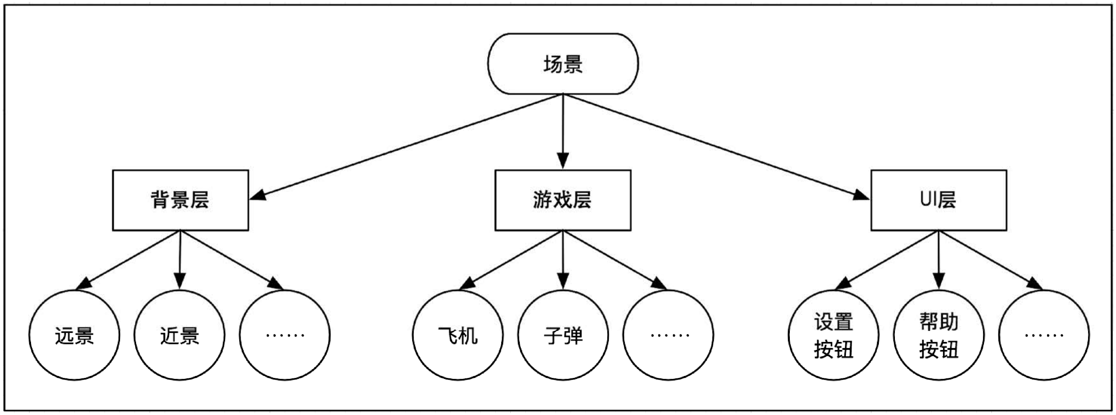


图3-8　Cocos游戏场景树

根节点为场景，场景的子节点是一些层，你应该合理地分类和管理好你的层，然后把对应的节点放在对应的层中。例如，背景层放一些背景相关的精灵，游戏层放英雄、敌人等游戏对象，而UI层则应该放一些按钮之类的UI。当然，你还可以创建更多的层来管理游戏。

3.8　标签

Label标签一般用来显示玩家的名字、等级、物品数量等文本。实际上，它和`cc.Sprite`等一样，属于引擎核心渲染部分。Label标签分为LabelTTF、LabelBMFont和LabelAtlas。

3.8.1　cc.LabelTTF 字体标签

TTF（TrueType Font）是一种字库规范，是 Apple 公司和 Microsoft 公司共同推出的字体文件格式。随着Windows的流行，TTF已经变成最常用的一种字体文件表示方式。LabelTTF直接使用TTF 字库，可以支持全部的中文，但是效率稍微低下。创建一个LabelTTF的代码如下：

```
1  var node = new cc.LabelTTF("Hello World", "Arial", 32);
```

其中，`Hello World`为文本内容，`Arial`为系统自带的字体库，也是默认字体库，32表示字体大小。另外，如果创建LabelTTF对象时未给出字体名字或者给出的名字系统中不存在，则使用引擎

默认字体初始化对象。cc.LabelTTF的构造函数以及参数说明如表3-17所示。

表3-17 **cc.LabelTTF**构造函数以及参数说明

构造函数	cc.LabelTTF(text, fontName, fontSize, dimensions, hAlignment, vAlignment)	
参数列表	参　数	参数说明
	font	字体文件或字体名称
	fontSize	字体大小，值必须大于0
	dimensions	文本的内容大小
	hAlignment	水平对齐方式
	vAlignment	垂直对齐方式

3

hAlignment可取的值如下。

- ❑ cc.TEXT_ALIGNMENT_LEFT：表示左对齐。
- ❑ cc.TEXT_ALIGNMENT_CENTER：表示居中对齐。
- ❑ cc.TEXT_ALIGNMENT_RIGHT：表示右对齐。

vAlignment可取的值如下。

- ❑ cc.VERTICAL_TEXT_ALIGNMENT_TOP：表示上对齐。
- ❑ cc.VERTICAL_TEXT_ALIGNMENT_CENTER：表示居中对齐。
- ❑ cc.VERTICAL_TEXT_ALIGNMENT_BOTTOM：表示下对齐。

此外，cc.LabelTTF常用API如表3-18所示。

表3-18 **cc.LabelTTF**常用API

返回值类型	参数类型	函　数	说　明
String	无	getString()	获取内容
无	String	setString(text)	设置内容
Number	无	getLineHeight()	获取行高
无	Number	setLineHeight(lineHeight)	设置行高
cc.Size	无	getFontSize()	获取字号大小
无	cc.Size	setFontSize(size)	设置字号大小
cc.Size	无	getDimensions()	获取文本内容大小
无	cc.Size|Number Number	setDimensions(dim, height)	设置文本内容大小
String	无	getFontName()	获取字体文件或名称
无	String	setFontName(name)	设置字体文件或名称
cc.FontDefinition	无	getTextDefinition()	获取文本属性定义
无	cc.FontDefinition	setTextDefinition(theDefinition)	设置文本属性定义
Number	无	getHorizontalAlignment()	获取文本水平对齐方式

（续）

返回值类型	参数类型	函　　数	说　　明
无	Number	setHorizontalAlignment(alignment)	设置文本水平对齐方式
Number	无	getVerticalAlignment()	获取文本垂直对齐方式
无	Number	setVerticalAlignment(alignment)	设置文本垂直对齐方式
无	cc.Color|Number cc.Size|Number Number null | Number	enableShadow(a, b, c, d)	启用文本阴影效果
无	无	disableShadow()	关闭文本阴影效果
无	cc.Color Number	enableStroke(strokeColor, strokeSize)	启用文本描边效果
无	无	disableStroke()	关闭文本描边效果

3.8.2　cc.LabelBMFont 位图标签

cc.LabelBMFont是一种纹理图集形式的标签类，支持FNT类型的文件，它使用图片文件显示文本内容，每个字符都可以被看做一个精灵，每个精灵“映射”纹理图集中的某一块区域。实际上，每次切换字符就是这个精灵的坐标变化。

位图字体由专门的编辑工具制作而成，例如Mac OS X下的Glyph Designer（下载地址：https://71squared.com/glyphdesigner）和Windows下的BMFont（下载地址：http://www.angelcode.com/products/bmfont/）等。通过编辑器制作完FNT字体文件后，会生成一个.png格式的文字集图片和一个.fnt格式的字体信息文件，其中.fnt文件中包含了对应文字集图片的名字、字符在文字集图片中的坐标、宽度以及高度等信息。

创建一个cc.LabelBMFont的代码如下：

```
1  var node = new cc.LabelBMFont("123", "res/unit03_base/fonts/number.fnt");
```

其中，123为文本内容，"res/unit03_base/fonts/number.fnt"为FNT字体文件的路径。cc.LabelBMFont构造函数以及参数说明如表3-19所示。

表3-19　cc.LabelBMFont的构造函数以及参数说明

构造函数	cc.LabelBMFont(str, fntFile, width, alignment, imageOffset)	
参数列表	参　　数	参数说明
	str	文本内容
	fntFile	字体文件
	width	最大线宽
	alignment	对齐方式（水平方向）
	imageOffset	第一个字符在位图文件中的偏移量

其中alignment可取的值如下。

- cc.TEXT_ALIGNMENT_LEFT：表示左对齐。
- cc.TEXT_ALIGNMENT_CENTER：表示居中对齐。
- cc.TEXT_ALIGNMENT_RIGHT：表示右对齐。

此外，cc.LabelBMFont常用API如表3-20所示。

表3-20 cc.LabelBMFont常用API

返回值类型	参数类型	函 数	说 明
String	无	getString()	获取内容
无	String	setString(text)	设置内容

3.8.3 cc.LabelAtlas 字符映射标签

cc.LabelAtlas号称速度最快的标签，它直接使用图片初始化文字对象，这一般适合用在频繁更新的标签上，例如精确到毫秒的定时器等。

cc.LabelAtlas将传入的文字集图片“拆分”成指定高度和宽度的小块，cc.LabelAtlas对文字集图片中内容的要求是非常严格的，即你不能有任何一个像素的偏差。图片中的文本要按照ASCII顺序排列，在创建cc.LabelAtlas标签的时候，需要设置开始字符。在这种情况下，文字集图片的设计自由度就不那么高了，你必须保证要使用的字符的ASCII是连续的。只有数字、字母以及一些运算符号符合这样的标准，也只有这些情况可以考虑使用LabelAtlas，这直接导致cc.LabelAtlas的使用不是那么广泛，更多的则是考虑TTF或BMFont实现。

cc.LabelAtlas标签的创建代码如下：

```
1  var node = new cc.LabelAtlas("789", "res/unit03_base/fonts/score.png", 41, 50, '0');
```

cc.LabelAtlas的构造函数以及参数说明如表3-21所示。

表3-21 cc.LabelAtlas构造函数以及参数说明

构造函数	cc.LabelAtlas(strText, charMapFile, itemWidth, itemHeight, startCharMap)	
参数列表	参 数	参数说明
	strText	文本内容
	charMapFile	图片名称
	itemWidth	每个“小块”的宽度
	itemHeight	每个“小块”的高度
	startCharMap	起始字符

此外，cc.LabelAtlas常用API如表3-22所示。

表3-22　`cc.LabelAtlas`常用API

返回值类型	参数类型	函　　数	说　　明
String	无	getString()	获取文本内容
无	String	setString(value)	设置文本内容

3.8.4　使用编辑器制作 FNT 字体

这里我以Windows下的BMFont编辑器为例，讲解一下如何制作自己的FNT字体。首先，我准备了10张小数字图片，每张的尺寸为40 × 50像素，如图3-9所示。



图3-9　10张数字小图

接着打开BMFont编辑器，页面如图3-10所示。

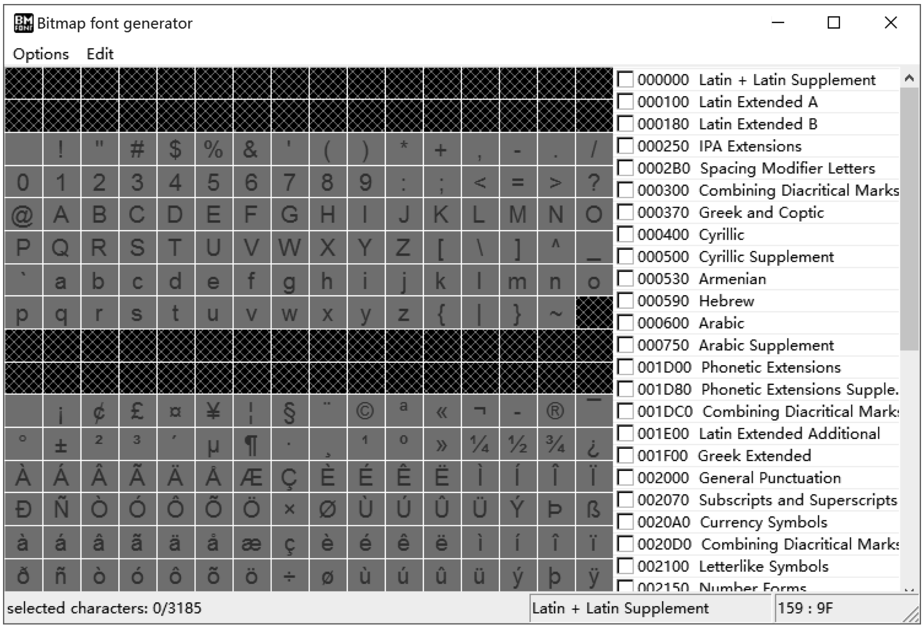


图3-10　BMFont编辑器

编辑器中左边为当前支持的字符，右边为字体库。点击Edit→Open Image Manager，打开Image Manager面板，如图3-11所示。

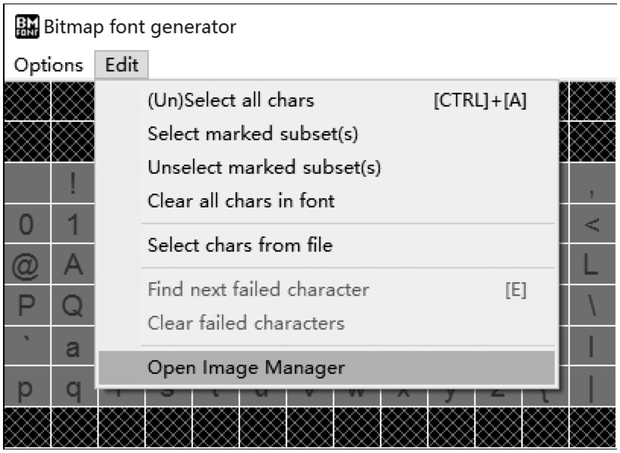


图3-11 打开Image Manager面板

在Image Manager面板中，点击Image→Import image...菜单，将事先准备好的文字图片导入进来，如图3-12所示。

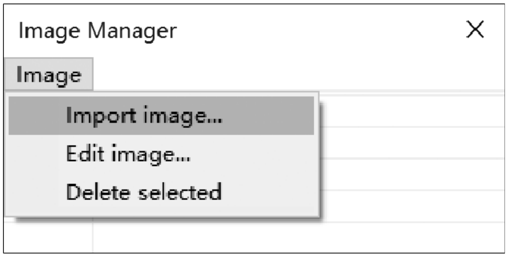


图3-12 在Image Manager中导入图片

此时先导入number_0.png，导入图片后，将出现如图3-13所示的面板。

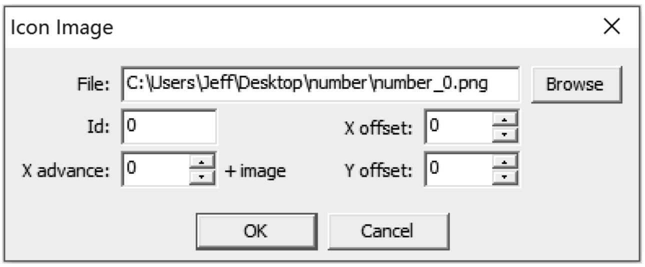


图3-13 Icon Image面板

其中，需要修改的是Id的值。当前导入的图片是number_0.png，在字符集中将鼠标放置在字符0的上方，Id值便在BMFont编辑器主页面中的右下角显示出来，如图3-14所示。此时得到0的Id值为48，那么将图3-13中的Id改为48，点击OK按钮即可。

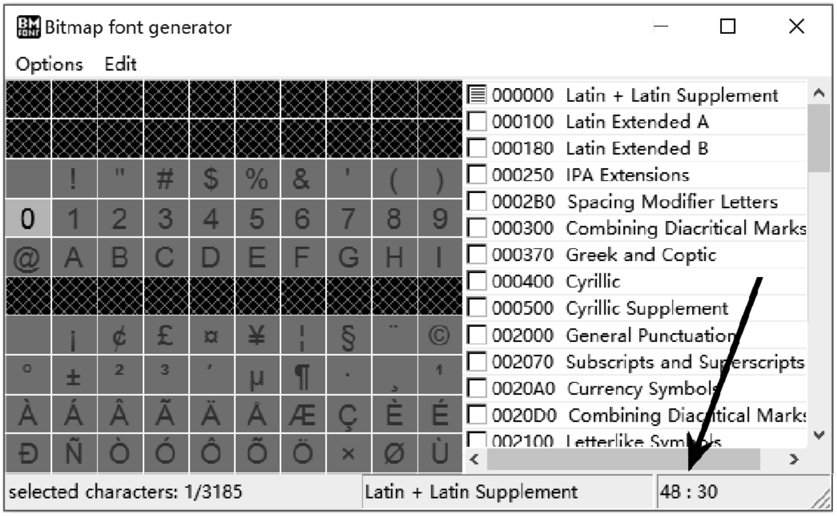


图3-14　获取Id值

重复此步骤，将1到9的图片全部导入进来。完成之后，BMFont编辑器中对应的字符的右下角会出现一个蓝色的小点，如图3-15所示。



图3-15　导入图片后的情况

接着，点击Options→Font settings，打开Font Settings面板，如图3-16所示，将Size设置为40像素，这是因为我准备的数字小图片的尺寸是40 × 50。最后，点击OK按钮关闭Font Settings面板。

完成上述所有步骤之后，便可将字体导出。点击Options→Export options，打开Export Options面板，如图3-17所示。将Spacing设为0，Bit depth选择32，Textures选择png格式，最后点击OK按钮，关闭Export Options面板。

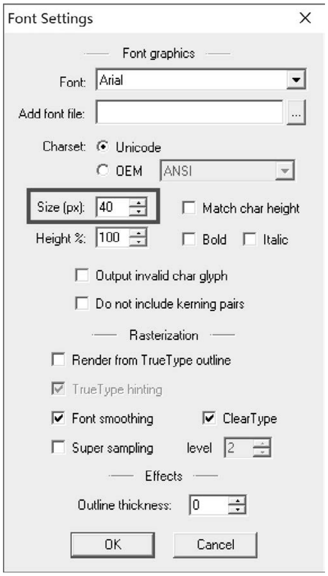


图3-16　Font Settings面板

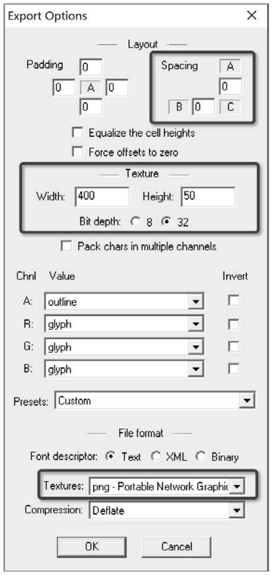


图3-17　Export Options面板

最后，点击Options→Save bitmap font as… 导出字体文件，可以得到一个.png和一个.fnt文件。

3.9　实例——《保卫萝卜 2》主页面开发

在上一章的实例中，你已经完成了《保卫萝卜2》项目的创建，结合本章所学的知识，开发《保卫萝卜2》的主页面已然不在话下。所以在本章的实例中，我将与你一起开发《保卫萝卜2》的主页面，完成后的效果如图3-18所示。



图3-18　《保卫萝卜2》主页面

3.9.1　主页面设计思路

在正式编写代码之前，你应该先看看《保卫萝卜2》主页面的美术素材提供情况，如表3-23所示。

表3-23　《保卫萝卜2》主页面图片资源表

<table>
<tr><td colspan="2"></td><td colspan="3"></td></tr>
<tr><td colspan="2">主页面背景图</td><td colspan="3">主页面前景图</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>“问号”按钮抬起状态</td><td>“问号”按钮按下状态</td><td>“设置”按钮抬起状态</td><td>“设置”按钮按下状态</td><td>萝卜</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>怪物1号</td><td>怪物2号</td><td>怪物3号</td><td>怪物4号</td><td>怪物5号</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>怪物6号</td><td>怪物6号的手</td><td>锁</td><td>云朵1</td><td>云朵2</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>云朵3</td><td>“开始冒险”按钮
抬起状态</td><td>“开始冒险”按钮
按下状态</td><td>“天天向上”按钮
抬起状态</td><td>“天天向上”按钮
按下状态</td></tr>
</table>

结合图3-9和表3-1，你应该会发现，《保卫萝卜2》的主页面开发并不难，可将页面的内容用两个层管理，一个是背景层，一个是用户交互层。

背景层仅放一张主页面背景图，其余的因为有层级关系以及用户交互，所以放到了用户交互层。

用户交互层主要是放了一个“开始冒险”和“天天向上”菜单，以及“帮助”和“设置”这两个按钮。另外，怪物、萝卜以及云朵因为有层级关系，所以，我不得不也将它们放到用户交互层当中来。那么接下来，就开始开发《保卫萝卜2》的游戏主页面吧。

3.9.2 视图设定

首先，在WebStorm中展开Carrot工程，打开index.html文件，删除body标签中的background属性，使body不带背景颜色，这么做只是为了让canvas在body中更明显地显示出来。然后，将canvas标签中的width和height属性的值改为1136和640，把视图的设计分辨率的尺寸大小也设置为宽1136、高640，它应该在main.js文件中的cc.game.onStart()方法中被调用到。代码如下：

```
1  //cc.view.setDesignResolutionSize(800, 450, cc.ResolutionPolicy.SHOW_ALL);
2  cc.view.setDesignResolutionSize(1136, 640, cc.ResolutionPolicy.SHOW_ALL);
```

这里分别将canvas和游戏的设计分辨率尺寸设置为1136 × 640，是因为《保卫萝卜2》的美术素材就是基于此分辨率设计的。cc.view.setDesignResolutionSize()函数用来设置游戏设计分辨率的尺寸以及适配策略，具体详见第7章。

3.9.3 代码文件架构

在src文件夹下删除app.js文件，并创建如图3-19所示的文件夹以及.js文件，然后在project.json中引入这些.js文件，如图3-20所示。

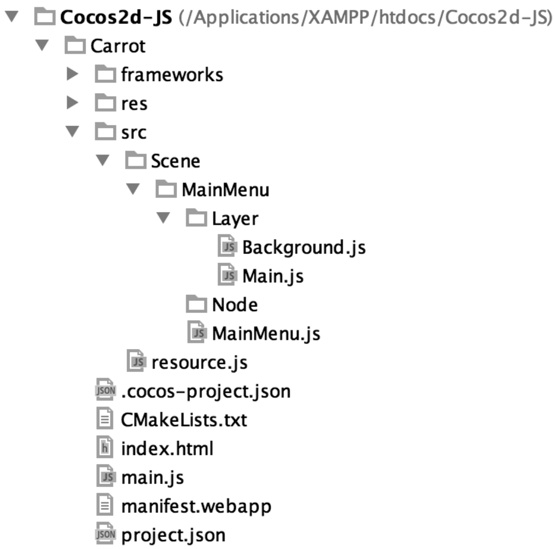


图3-19 Cocos游戏代码文件组织结构

```
"jsList" : [
    "src/resource.js",

    "src/config.js",
    "src/Scene/MainMenu/Layer/Background.js",
    "src/Scene/MainMenu/Layer/Main.js",
    "src/Scene/MainMenu/MainMenu.js"
]
```

图3-20　jsList数组

说明　以后再新建的.js文件，都默认需要在project.json的jsList数组中引入，本书不再一一强调。

图3-19是我个人组织场景的一种方式，我总喜欢把场景元素尽可能地按功能、模块组织好，使得代码条理清晰、场景组织井然有序。

在图3-19中，MainMenu文件夹为一个模块，用来存放主页面中的相关实现代码。MainMenu.js文件为主页面场景实现，Layer文件夹下的Background.js和Main.js分别管理主页面场景中的背景层以及用户交互层，一般我也会将游戏逻辑放在Main.js中。

3.9.4　主页面 BackgroundLayer 开发

当相关.js文件创建完毕之后，便可以开发Carrot的游戏场景了。此时在res路径下新建MainMenu文件夹，然后打开随书资源，把主页面的美术素材复制到MainMenu文件夹中，接着在resource.js文件中声明这些文件。

说明　在后续章节中，资源声明或加载等步骤不再阐述，读者可自行查阅随书源码中的resource.js文件。

注意　在Web上，资源要预加载，即需要在resource.js中定义，否则用此资源创建出来的精灵的大小都是0。

接着，在Scene/MainMenu/Layer/Background.js文件中键入如下代码：

```
1  var MMBackgroundLayer = cc.Layer.extend({
2      ctor : function () {
3          this._super();
4          // 加载[背景]
5          this.loadBackgound();
6          return true;
7      },
8      loadBackgound : function(){
9          var node = new cc.Sprite("res/MainMenu/zh/front_bg.png");
```

```
10          this.addChild(node);
11          node.setPosition(cc.winSize.width / 2, cc.winSize.height / 2);
12      }
13 });
```

第1行代码定义了MMBackgroundLayer类，它继承了cc.Layer。类名MMBackgroundLayer最前面的MM为MainMenu的缩写，这也是我个人的一个习惯，主要是为了避免和其他场景中的BackgroundLayer混淆。第2行代码定义了MMBackgroundLayer类的构造函数，通过this._super()接口调用了父类的ctor构造函数。

第8行到第12行是加载背景精灵的函数，它在构造函数中被调用到。在第11行代码中，将背景精灵的坐标设置为屏幕的正中间。

3.9.5 主页面 MainLayer 开发

使用同样的方式，在Scene/MainMenu/Layer/Main.js文件中定义MMMainLyer。MMMainLayer的开发比MMBackgroundLayer复杂一些，所以这里我把MMMainLayer中要做的事情用注释的方式写在了构造函数中，然后逐个开发，具体如下：

```
1  var MMMainLayer = cc.Layer.extend({
2      ctor : function(){
3          this._super();
4          // 加载[“开始冒险”和“天天向上”菜单]
5          // 加载[“设置”按钮]
6          // 加载[“帮助”按钮]
7          // 加载底部的1号和3号[怪物]
8          // 加载底部遮挡在1号、5号以及6号怪物之前的[云朵]
9          // 加载前面的5号[怪物]
10         // 加载前面遮罩在5号怪物身上的[云朵]
11         // 加载[萝卜]
12         // 加载[前景]
13     }
14 });
```

毫无疑问，秉承模块化开发的风格，我会对这些要做的事情都写一个函数。下面先从加载菜单开始，它的实现代码如下：

```
1  // 加载[“开始冒险”和“天天向上”菜单]
2  loadMenu : function(){
3      // 开始冒险
4      var startNormal    = new cc.Sprite("res/MainMenu/zh/front_btn_start_normal.
       png");
5      var startPress   = new cc.Sprite("res/MainMenu/zh/front_btn_start_pressed.
       png");
6      var startDisabled  = new cc.Sprite("res/MainMenu/zh/front_btn_start_normal.
       png");
7      var start = new cc.MenuItemSprite(
8          startNormal,
9          startPress,
10         startDisabled,
```

```
11            function(){
12                cc.log("点击“开始冒险”按钮");
13            }.bind(this));
14            start.setPosition(cc.winSize.width / 2-8, cc.winSize.height / 2+75);
15
16        //“天天向上”菜单项的开发与开始冒险类似，此处省略，具体见随书代码
17        // ......
18
19        var menu = new cc.Menu(start, floor);
20        this.addChild(menu);
21        menu.setPosition(0, 0);
22    }
```

第7行代码创建了开始冒险的精灵菜单项，其中需要3个精灵来表示正常、按下、禁用状态。第11行代码为此菜单项点击之后所触发的函数，最后在第18行将菜单项添加到菜单中。

而其他的怪物、萝卜、云朵等都只是一个精灵，创建方式较为简单。下面以创建萝卜为例，相关代码如下：

```
1  // 加载[萝卜]
2  loadCarrot : function(){
3      var node = new cc.Sprite("res/MainMenu/front_carrot.png");
4      this.addChild(node);
5      node.setPosition(cc.winSize.width / 2 + 100, 20);
6  }
```

因为其他怪物和云朵等都是简单的精灵，创建代码也都差不多，所以这里就不贴出全部代码，读者可自行查阅随书代码。另外，因为“设置”“帮助”按钮属于UI控件（参见第10章），所以，它们的正确开发方式将在第10章中优化，这里我先用精灵的方式实现。

当主页面都开发完毕之后，运行项目，效果如图3-18所示。

3.10　小结

通过本章的学习，我们知道在Cocos2d-JS游戏引擎中存在导演对象`cc.director`，它是整个游戏的统筹，负责设置游戏的运行环境，把控整个游戏的主循环以及游戏场景的切换等工作。此外，我们掌握了Cocos引擎的三大元素——场景、层和精灵，它们都继承自`cc.Node`节点类，而`cc.Node`为子类提供了大量的属性和API。场景类似京剧中的舞台，一次只能运行一个，每个Cocos2d-JS游戏至少存在一个以上的场景。层对应京剧中的幕布，可将游戏中的节点按功能划分到对应的层。而精灵可以理解为京剧中的演员，在2D游戏中，你可以将精灵直接理解为一张图片。

3.11　参考资源

本章的参考资源为Cocos2d-JS 官方API： http://www.cocos.com/doc/jsdoc/index.html。

第 4 章 动作模块

在一些回合制游戏中，你可能会看到这样一个画面：某个游戏人物从它当前的位置快速移动到敌方阵地，攻击了一下敌方之后又立马跑回来，这样的指令实际上是一个组合动作。除此之外，例如人物的奔跑、网球的曲线运动、心脏跳动、落叶飘落等效果都可以通过动作实现。由此可见，动作是游戏中极其重要的一个模块，熟练掌握Cocos2d-JS的动作，可以让你的游戏锦上添花。本章将围绕Cocos2d-JS的动作展开，并让《保卫萝卜2》的主页面动起来。

本章内容：

- cc.Action
- 瞬时动作
- 持续动作
- 变速动作
- 实例——让《保卫萝卜2》的主界面动起来

4.1 cc.Action

动作作用于`cc.Node`，每个节点都可以通过`runAction(action)`函数运行一个或多个动作，每个动作保存着指定的规则供节点运作。在Cocos2d-JS中，动作对应的类是`cc.Action`，它是所有动作类的父类，其构造函数如下：

```
1  cc.Action = cc.Class.extend({
2      originalTarget:null,
3      target:null,
4      tag:cc.ACTION_TAG_INVALID,
5      ctor:function () {
6          this.originalTarget = null;
7          this.target = null;
8          this.tag = cc.ACTION_TAG_INVALID;
9      }
10     //......
11 });
```

可以看到，该类定义了3个属性：初始目标（`originalTarget`）、目标（`target`）以及动

作标签（tag）。动作是允许克隆的，originalTarget始终指向第一次运行此动作的节点，而target则为当前正在运行此动作的节点，tag则是一个动作的唯一标识，它的默认值为–1。tag一般被用在cc.Node的stopActionByTag(tag)等函数中。

动作类的大多数实现类都继承有限时间类（cc.FiniteTimeAction），cc.FiniteTimeAction类定义了reverse()函数，调用此函数可以获得一个与当前动作相反的动作，这称为逆动作。例如，一个精灵运行了旋转负45°的动作，那么调用reverse()函数，将返回一个旋转正45°的动作，但是并不是所有的动作都可返回逆动作，详情可参见对应动作的源码。另外，cc.FiniteTimeAction还派生出瞬时动作cc.ActionInstant和持续性动作cc.Action-Interval，它们各自的子类继承关系如图4-1所示。

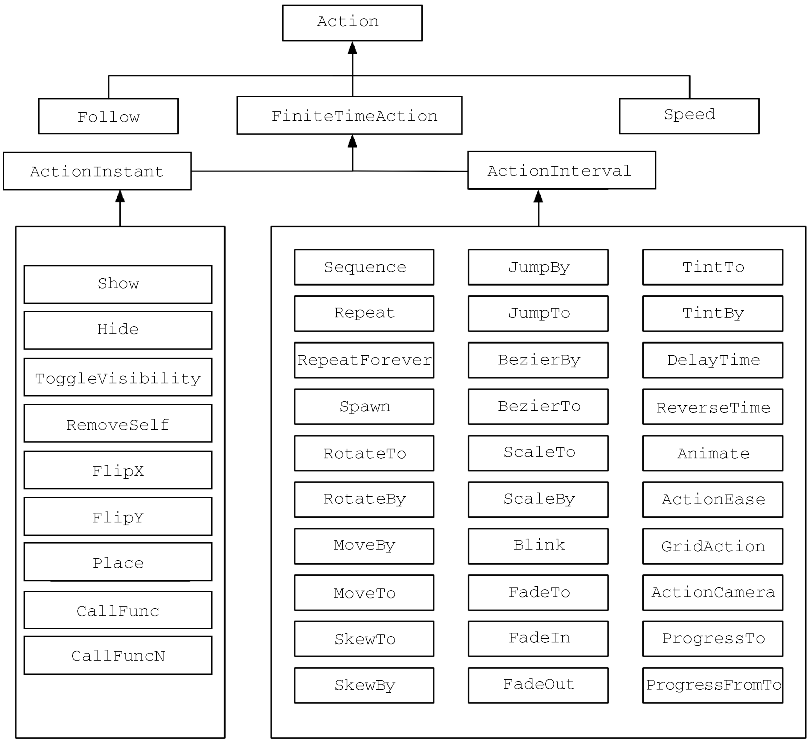


图4-1　动作继承关系

4.2　瞬时动作

在一帧内执行结束的动作称为瞬时动作。cc.ActionInstant是所有瞬时动作类的父类，例如cc.Place、cc.Hide、cc.RemoveSelf等都是瞬时动作。

4.2.1 cc.Place

cc.Place的作用是将节点放置到某个指定位置，实际上它只是修改了节点的position坐标。所以，若你想通过一个动作将节点放置到屏幕坐标(100, 86)，可以像下面这样做：

```
1  var place = cc.place(100, 86);
2  node.runAction(place);
```

说明 Cocos2d-JS简化了动作的创建方式，为所有的动作定义了快速创建的函数。例如，cc.place函数的实现如下：

```
1 cc.place = function (pos, y) {
2     return new cc.Place(pos, y);
3 };
```


4.2.2 cc.FlipX 和 cc.FlipY

cc.FlipX和cc.FlipY这两个动作在本质上是一样的，只是作用的方向不同。cc.FlipX是将目标水平翻转，而cc.FlipY则是将目标垂直翻转，true和false表示是否翻转。一般情况下，cc.FlipX和cc.FlipY作用于cc.Sprite或cc.Sprite的子类上，代码如下：

```
1  var flipX = cc.flipX(true);
2  var flipY = cc.flipY(false);
```

4.2.3 cc.Show 和 cc.Hide

几乎所有即时动作的实现都是直接修改其对应的属性，cc.Show和cc.Hide也不例外。cc.Show和cc.Hide通过修改节点的visible属性来达到显示和隐藏的功能。通过下列代码可以创建一个显示或隐藏的动作：

```
1  var show = cc.show();   // 创建一个[显示]动作
2  var hide = cc.hide();   // 创建一个[隐藏]动作
```

4.2.4 cc.ToggleVisibility

cc.ToggleVisibility动作可以切换节点的可视（Visible）属性，当节点可见时，运行此动作，则节点会被隐藏，反之同理。创建cc.ToggleVisibility动作的代码如下：

```
1  var toggleVisibility = cc.toggleVisibility();
```

4.2.5 cc.RemoveSelf

顾名思义，cc.RemoveSelf就是将正在运行此动作的节点从父节点移除，它实际上是调用了target（正在运行此动作的节点）的removeFromParent(isNeedCleanUp)函数。你可以在

cc.RemoveSelf的update函数中看到如下代码：

```
1  update:function(dt){
2      this.target.removeFromParent(this._isNeedCleanUp);
3  }
```

cc.RemoveSelf类中声明了一个属性_isNeedCleanUp，该属性默认为true，表示目标节点以及它所关联的动作、调度器等都将被移除。若_isNeedCleanUp为false，则只移除目标节点。

cc.RemoveSelf的构造函数是一个有参构造函数，因为_isNeedCleanUp的默认值为true，所以在实际开发中可以省略这个参数，相关代码如下：

```
1  var moveSelf = cc.removeSelf(); // 等同于cc.removeSelf(true)
2  var moveSelf = cc.removeSelf(false);
```

4.2.6　cc.CallFun

cc.CallFunc是即时动作中较为特殊的一种，它将一个函数包装成动作，当节点运行此动作的时候，便会回调此函数。通常情况下，cc.CallFun会结合复合动作使用。例如，主角2秒内移动到屏幕(100, 200)后，说了一句："我已到达指定地点。"那么，这个功能的实现代码如下：

```
1  var moveTo = cc.moveTo(2, cc.p(100, 200));
2  var callback = cc.callFunc(function(){
3      cc.log("我已到达指定地点。");
4  }.bind(this));
5  node.runAction(cc.sequence(moveTo, callback));
```

说明　瞬时动作的实现比较简单，若读者感兴趣，可自行查阅各类动作的update函数。瞬时动作一般配合复合动作使用。

4.3　持续动作

需要持续一定时间才能完成的动作称为持续动作，例如花2秒的时间让精灵移动到屏幕坐标(300, 200)的位置。由于持续动作需要持续一段时间才能完成，所以所有的持续动作都需要接收一个用于控制动作执行时间的参数duration。

大部分的持续动作可以分为两种——xxTo和xxBy，前者表示最终值，而后者表示相对改变值。例如，用1秒的时间将节点移动到屏幕坐标(–20, 50)的位置，则代码如下面的第1行所示；若让节点相对当前坐标进行X轴偏移–20，Y轴偏移50，则代码应该如下面第2行所示：

```
1  var moveTo = cc.moveTo(1, cc.p(-20, 50));
2  var moveBy = cc.moveBy(1, cc.p(-20, 50));
```

根据持续动作的作用不同，我们将持续动作分为属性变化动作和视觉特效动作两大类。

4.3.1 属性变化动作

属性变化动作，即运行动作的过程中会改变节点对应属性的动作，它们大概有如下几种。

1. cc.MoveTo和cc.MoveBy

Move动作使节点做直线运动，通常用来移动游戏内的角色等，其函数原型如下：

```
1  cc.moveTo(duration, position, y);
2  cc.moveBy(duration, position, y);
```

其中，参数duration为持续时间，position可以为cc.p(x,y)类型的值，也可以为坐标的x值。当position为cc.p(x, y)时，y可以省略。示例如下：

```
1  // 用时1秒，将节点移动到屏幕(128, 80)的位置
2  var moveTo = cc.moveTo(1, cc.p(125, 80)); // 等同于cc.moveTo(1, 125, 80);
3  // 用时1秒，节点相对当前坐标，X轴向左移动88，Y轴向上移动99
4  var moveBy = cc.moveBy(1, cc.p(-88, 99)); // 等同于cc.moveBy(1, -88, 99);
```

2. cc.JumpTo和cc.JumpBy

Jump动作以一定的轨迹让节点跳跃到指定的位置，其API如下：

```
1  cc.jumpTo(duration, position, y, height, jumps);
2  cc.jumpBy(duration, position, y, height, jumps);
```

可以发现，形参中也有position，这意味着若传入进去的参数为cc.p(x, y)类型的参数，则y同样可以省略。height为跳跃高度，jumps为跳跃的次数。示例如下：

```
1  //【举例】：1秒时间，跳到(100, 86)，跳跃高度为50像素，总共跳跃4次
2  var jumpTo = cc.jumpTo(1, cc.p(100, 86), 50, 4);
3  //【举例】：1秒时间，原地跳，跳跃高度为100像素，总共跳跃4次
4  var jumpBy = cc.jumpBy(1, cc.p(0, 0), 100, 4);
5  node.runAction(cc.sequence(jumpTo, jumpBy));
```

说明 cc.sequence动作将一组动作包装成一个序列，然后挨个执行，具体见4.3节。

3. cc.BezierTo和cc.BezierBy

Bezier动作可以让节点做曲线运动，而曲线则由贝塞尔曲线描述。贝塞尔曲线又称贝兹曲线或贝济埃曲线，是应用于二维图形应用程序的数学曲线。

每条贝塞尔曲线都包含一个起点和一个终点。在一条曲线中，起点和终点各自包含一个控制点，而控制点到起点（或终点）的连线称作控制线，所以每条贝塞尔曲线应该有两条控制线和两个控制点。控制点和起点（或终点）的角度以及长度决定了曲线的形状，如图4-2所示。

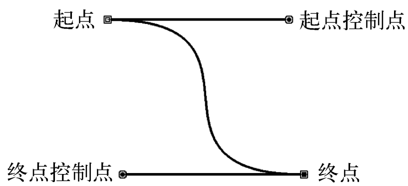


图4-2 贝塞尔曲线

Bezier动作的API如下：

```
1  cc.bezierTo(duration, control);
2  cc.bezierBy(duration, control);
```

其中，control参数为贝塞尔曲线的描述信息，它接收三个点坐标，分别是起点控制点、终点控制点、终点。所以，当你需要创建一个Bezier动作时，应该先配置好贝塞尔曲线的描述信息，就像下面这样：

```
1  var size = cc.winSize;
2  //  【举例】：
3  // 条件：当前节点坐标为cc.p(0, 0)
4  // 要求：用1秒时间做贝塞尔曲线运动，将节点从当前位置移动到屏幕右下角
5  var bezierToConfig = [
6      cc.p(0, size.height),             // 起点控制点
7      cc.p(size.width, size.height),    // 终点控制点
8      cc.p(size.width, 0)               // 终点
9  ];
10 var bezierTo = cc.bezierTo(1, bezierToConfig);
11
12 //【举例】：
13 // 条件：当前节点坐标为cc.p(cc.winSize.width, 0)
14 // 要求：用1秒时间做相对贝塞尔曲线运动，将X轴往负方向移动到屏幕宽度的一半
15 var bezierByConfig = [
16     cc.p(0, size.height),
17     cc.p(-size.width / 2, size.height),
18     cc.p(-size.width / 2, 0)
19 ];
20 var bezierBy = cc.bezierBy(1, bezierByConfig);
21 node.runAction(cc.sequence(bezierTo, bezierBy));
```

实际上，在Adobe Photoshop中，钢笔工具就是贝塞尔曲线的应用。所以，在开发中，我们可以利用Adobe Photoshop的钢笔工具调出贝塞尔曲线，然后将参数搬到代码中，即可得到你想要的Bezier动作。

4. cc.ScaleTo和cc.ScaleBy

Scale动作可以让节点在指定的时间内进行缩放，其API如下：

```
1  cc.scaleTo(duration, sx, sy);
2  cc.scaleBy(duration, sx, sy);
```

可以发现，除了duration参数之外，还有sx和sy这两个参数，它们分别表示X轴和Y轴上的缩放。但是在实际开发中，更多的需求是对整个节点进行缩放，所以，你只需指定一个整体缩放系数即可。这使得sy参数必定成为一个可以省略的参数，当sy省略时，cc.ScaleTo或cc.ScaleBy的_endScaleY属性会被强制设置为sx的值，这样便可实现X轴和Y轴的缩放比例一致。

下面通过一个示例来学习下Scale动作的用法，代码如下：

```
1  //【举例】：用1秒的时间，把图片缩放到【原始】大小的50%
2  var scaleTo = cc.scaleTo(1, 0.5);
```

```
3 //【举例】：用1秒的时间，把图片缩放到【当前】大小的200%
4 var scaleBy = cc.scaleBy(1, 2);
5 //【举例】：用1秒的时间，把x方向缩放到【原始】大小的50%， y方向缩放到【原始】大小的150%
6 var scale = cc.scaleBy(1, 0.5, 1.5);
7 var scaleReverse = scale.reverse();
8 node.runAction(cc.sequence(scaleTo, scaleBy, scale, scaleReverse));
```

5. cc.RotateTo和cc.RotateBy

Rotate动作用于旋转节点，其API如下：

```
1 cc.rotateTo(duration, deltaAngleX, deltaAngleY);
2 cc.rotateBy(duration, deltaAngleX, deltaAngleY);
```


和Scale类似，cc.rotateTo（或cc.rotateBy）的deltaAngleY也是可省略的。下面通过几个例子来介绍一下Rotate动作的用法，代码如下：

```
1 //【举例】：用1秒的时间，将X轴旋转到-90度，Y轴旋转到-45度
2 var rotate = cc.rotateTo(1, -90, -45);
3 //【举例】：用1秒的时间，整体旋转到90度
4 var rotateTo = cc.rotateTo(1, 90);
5 //【举例】：用1秒的时间，基于【当前】的角度整体旋转-90度
6 var rotateBy = cc.rotateBy(1, -90);
```

值得注意的是，在Cocos2d-JS中，垂直X轴向上表示0度，垂直Y轴向右表示90度。

4.3.2 视觉特效动作

视觉特效动作可以使节点实现一些视觉效果，常见的视觉效果有淡入、淡出、闪烁、帧动画等。

1. cc.FadeIn、cc.FadeOut和cc.FadeTo

cc.FadeIn和cc.FadeOut用来实现节点的淡入淡出效果，cc.FadeTo实现节点的透明度变化，其API如下：

```
1 cc.fadeIn(duration);
2 cc.fadeOut(duration);
3 cc.fadeTo(duration, opacity);
```

示例代码如下：

```
1 //【举例】：用1秒的时间淡出
2 var fadeOut = cc.fadeOut(1);
3 //【举例】：用1秒的时间淡入
4 var fadeIn = cc.fadeIn(1);
5 //【举例】：用1秒的时间将透明度渐变到半透明（128）
6 var fadeTo = cc.fadeTo(1, 128);
7 node.runAction(cc.sequence(fadeOut, fadeIn, fadeTo));
```

2. cc.TintTo和cc.TintBy

Tint动作使节点发生色调变化，在实际开发中，此动作比较少用，其API和示例如下：

```
1  cc.tintTo(duration, deltaRed, deltaGreen, deltaBlue);
2  cc.tintBy(duration, deltaRed, deltaGreen, deltaBlue);
```

其中deltaRed、deltaGreen和deltaBlue分别表示红色分量、绿色分量以及蓝色分量。示例如下：

```
1  //【举例】：用1秒的时间，红色和蓝色分量升到255，绿色分量降到0
2  var tintTo = cc.tintTo(1, 255, 255, 0);
3  //【举例】：用1秒的时间，基于【当前】的颜色分量值，红色和蓝色分量不变，绿色分量上升255
4  var tintBy = cc.tintBy(1, 0, 0, 255);
5  node.runAction(cc.sequence(tintTo, tintBy));
6  var fadeTo = cc.fadeTo(1, 128);
7  node.runAction(cc.sequence(fadeOut, fadeIn, fadeTo));
```

3. cc.Blink

Blink动作使节点发生闪烁，例如人物掉血闪烁效果就是基于此动作实现的，其API如下：

```
1  cc.blink(duration, blinks);
```

示例如下：

```
1  //【举例】：1秒时间，闪烁10次
2  var actionBlink = cc.blink(1, 10);
3  node.runAction(actionBlink);
```

4. cc.Animation

实际上，精灵可以实现帧动画效果。帧动画的原理是将一组连续的动画分解成帧，每一帧就是一张纹理，然后通过指定的时间间隔将这些动画连续播放出来，从而形成帧动画，这类似于胶片电影。

在Cocos2d-JS中，帧动画用cc.Animate表示，其创建方式如下：

```
1  var animation = new cc.Animation(frames, delay, loops);
```

其中，frames是一个数组，里面存放着所有要播放的帧，delay为每帧播放的时间间隔，loops为帧动画循环播放次数。不过，这里的参数也都是可以省略的。

帧动画的实现又分为两种，其一为直接通过图片文件的方式，其二为plist精灵表单（制作方式见第11章）。通过图片文件直接创建的方式如下：

```
1  var animation = new cc.Animation();        // 创建动画
2  animation.addSpriteFrameWithFile(res.sh_node_64_png);
3  animation.addSpriteFrameWithFile(res.sh_node_128_png);
4  animation.addSpriteFrameWithFile(res.sh_node_256_png);
5  animation.addSpriteFrameWithFile(res.sh_node_512_png);
6  animation.setDelayPerUnit(0.15);           // 设置[帧间隔]
7  animation.setRestoreOriginalFrame(true);   // 设置[是否恢复到第一帧]
8
9  var animate = cc.animate(animation);  // 通过cc.animate将animation包装成动作
10 node.runAction(animate);
```

上述代码第1行创建了一个不带参数的cc.Animation，第2行到第5行代码通过

addSpriteFrameWithFile方法添加了4帧动画。第6行代码设置帧动画间隔为0.15秒，第7行代码设置动画播放结束后回到第一帧。最后，第9行代码通过cc.animate函数将animation对象包装成动作。

在实际开发中，很少直接通过图片文件的方式创建帧动画。首先，直接通过图片文件创建的方式占用的内存较大（见第11章），其次零散的帧较难维护。所以，对于一般帧动画，我们更多会考虑使用plist精灵表单的方式创建，相关代码如下：


```
1  cc.spriteFrameCache.addSpriteFrames(res.u5_dance_plist, res.u5_dance_png);
                                                         // 添加帧缓存
2  var node = new cc.Sprite("#dance_0.png"); // 创建节点
3  this.addChild(node);
4  node.setPosition(568, 320);
5  var frames = [];
6  for (var i = 0; i < 13; i++) {
7      var str = "dance_" + i + ".png";  // 注意：这里不需要加#号
8      var frame = cc.spriteFrameCache.getSpriteFrame(str);
9      frames.push(frame);
10 }
11 var animation = new cc.Animation(frames, 0.15);
12 animation.setRestoreOriginalFrame(true);   // 设置[是否恢复到第一帧]
13 var animate = cc.animate(animation);    // 用cc.animate将animation包装成动作
14 node.runAction(animate);
```

首先，你需要像上述代码的第1行那样，将plist文件以及plist文件所对应的图片加载到帧缓存中，然后根据帧创建精灵。从第5行代码开始配置frames数组，其余部分与直接通过图片文件创建的方式一致。

4.3.3　复合动作

复合动作将各种普通动作组合起来，产生更为复杂多变的效果。例如，心脏跳动效果，就是多个Scale动作的组合。复合动作也是一个动作，所以同样通过node.runAction的方式执行。

1. cc.DelayTime

cc.DelayTime为延时动作，类似多线程中的睡眠，在这一段时间中，它什么也不做，只是起到延时作用。

另外，更为准确地说，cc.DelayTime动作并不是复合动作，但是在开发中，它只能结合复合动作使用。cc.DelayTime的API如下：

```
1  cc.delayTime(duration);
```

下面举个例子：

```
1  //【举例】：延时0.5秒，将节点的X轴坐标左移100像素
2  var delay = cc.delayTime(0.5);
3  var moveBy = cc.moveBy(1, cc.p(-100, 0));
4  var sequence = cc.sequence(delay, moveBy);
5  node.runAction(sequence);
```

2. cc.Repeat和cc.RepeatForever

cc.Repeat将动作重复执行指定的次数，而cc.RepeatForever则表示将动作无限重复执行，示例如下：

```
1  //【举例】：用时0.5秒，节点旋转-90度，重复4次
2  var rotate = cc.rotateBy(0.5, -90);
3  var repeat = rotate.repeat(4); // 重复4次
4  node.runAction(repeat);
5  //【举例】：用时0.5秒，节点旋转-90度，一直重复执行
6  var rotate = cc.rotateBy(0.5, -90);
7  var repeat = rotate.repeatForever(); // 一直重复
8  node.runAction(repeat);
```

3. cc.Sequence

cc.Sequence是一个序列动作，其作用是将一组普通动作包装成一个序列，然后按照顺序一个个执行。所以，cc.Sequence的动作是有先后顺序的，其API如下：

```
1  cc.sequence(action1, action2, action3, ...);
```

可以看出，cc.sequence的参数是个不定参数，示例如下：

```
1  //【举例】：每个动作耗时0.5秒，先后按顺序执行了移动、闪烁、缩放和旋转4个动作
2  var moveBy = cc.moveBy(0.5, cc.p(-100, 0));
3  var blink = cc.blink(0.5, 8);
4  var scale = cc.scaleTo(0.5, 1.5);
5  var rotate = cc.rotateTo(0.5, 90);
6  var sequence = cc.sequence(blink, moveBy, scale, rotate);
7  node.runAction(sequence);
```

4. cc.Spawn

cc.Spawn是一个并行动作，即同时并发执行所有被包含在cc.Spawn里面的动作，所以，cc.Spawn里面包含的动作不用像cc.Sequence那样注意顺序，而且可以随意放置。其API如下：

```
1  cc.spawn(action1, action2, action3, ...);
```

下面依旧举一个例子，让你更容易理解：

```
1  //【举例】：两个动作并发执行。其一：用1秒的时间，节点的X轴坐标左移200。其二：用时0.5秒，闪烁8次
2  var moveBy = cc.moveBy(1, cc.p(-200, 0));
3  var blink = cc.blink(0.5, 8);
4  var spawn = cc.spawn(blink, moveBy);
5  node.runAction(spawn);
```

4.4　变速动作

变速动作可以让指定的持续动作发生速度上的变化，分为两种——线性变化动作cc.Speed和非线性变化动作cc.ActionEase。

4.4.1　cc.Speed

cc.Speed为线性变速动作，其API如下：

```
1  cc.speed(action, speed);
```

其中action为要发生变速的动作，speed为发生变速的倍数。下面举一个例子：

```
1  //【举例】：用时0.5秒，节点旋转-90度，重复4次，速度为原来的5倍
2  var rotate = cc.rotateBy(0.5, -90);
3  var repeat = rotate.repeatForever();
4  var speed = cc.speed(repeat, 5);
5  node.runAction(speed);
```

4.4.2　cc.ActionEase

cc.ActionEase使动作发生非线性速度变化。cc.ActionEase动作分为5类：指数缓冲、Sine缓冲、弹性缓冲、跳跃缓冲和回震缓冲。每类又可细分为In、Out和InOut三种，所以cc.ActionEase动作总共为15个。

cc.ActionEase动作使用起来比较简单，下面以ElasticIn为例进行介绍：

```
1  //【举例】：用2秒时间，携带弹性缓冲效果，让节点的X轴坐标向左移动300像素
2  var moveTo = cc.moveBy(2, cc.p(300, 0));
3  var elasticInMoveTo = moveTo.easing(cc.easeElasticIn());
4  node.runAction(elasticInMoveTo);
```

所有cc.ActionEase动作的实现都类似上述代码的第三行，其格式为action.easing(cc.easeXXX())。图4-3为Cocos2d-JS各种变速动作曲线示意图。

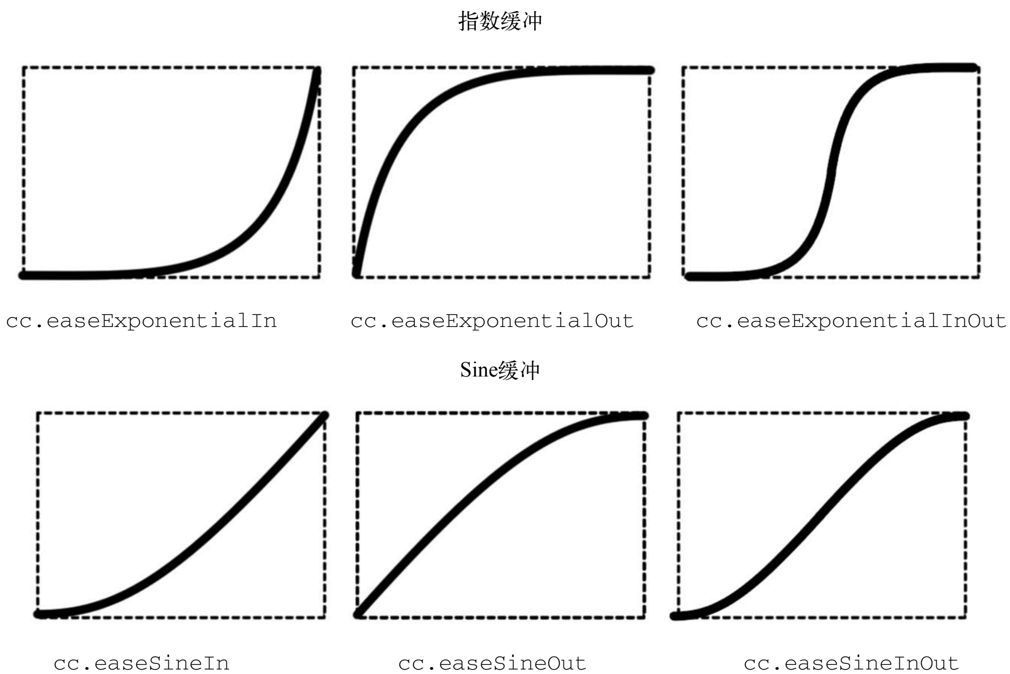


图4-3　Cocos2d-JS各种变速动作曲线示意图

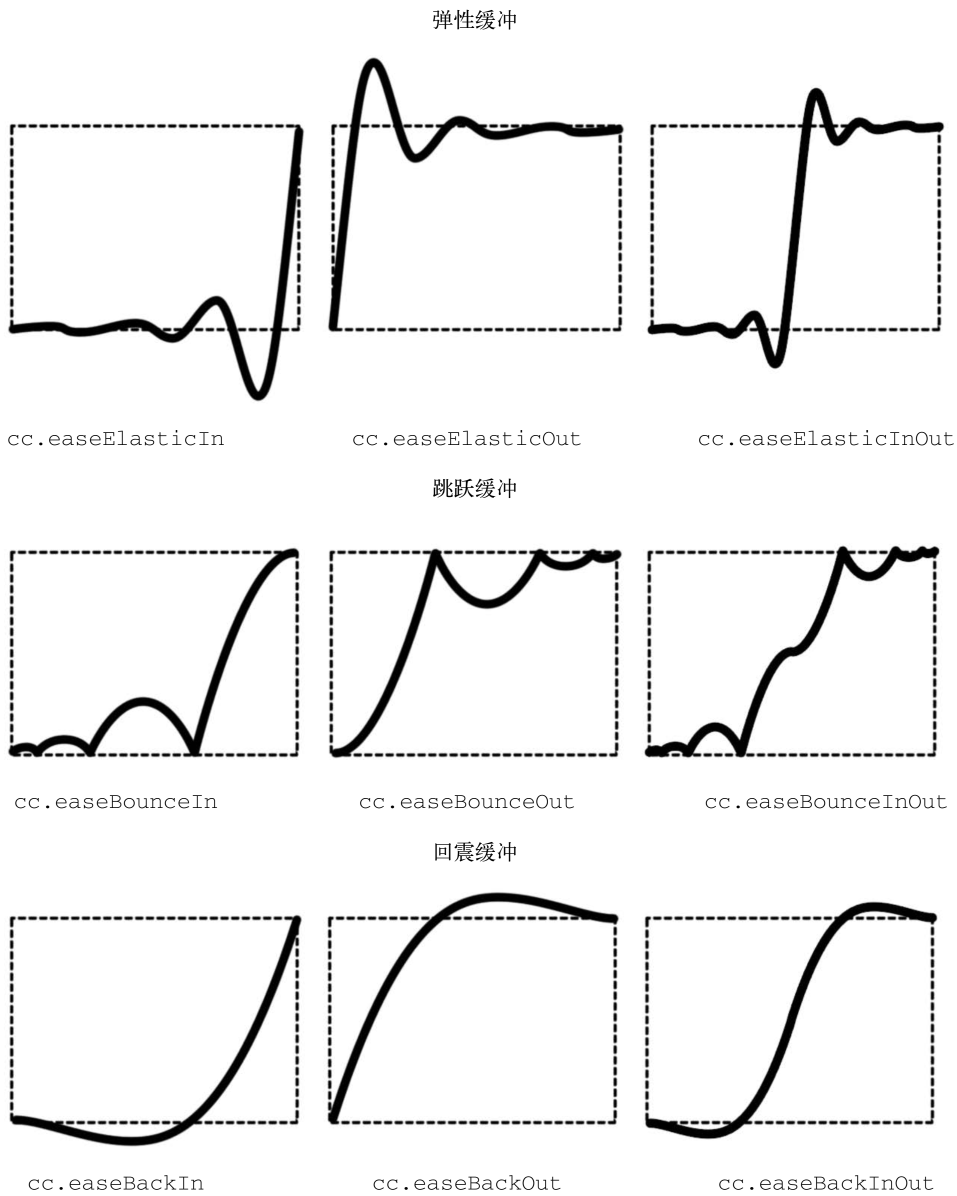

弹性缓冲
cc.easeElasticIn
cc.easeElasticOut
cc.easeElasticInOut
跳跃缓冲
cc.easeBounceIn
cc.easeBounceOut
cc.easeBounceInOut
回震缓冲
cc.easeBackIn
cc.easeBackOut
cc.easeBackInOut

图4-3 （续）

4.5　实例——让《保卫萝卜2》的主界面动起来

在上一章中，我们创建完《保卫萝卜2》主页面上的所有节点，并且正确摆放好了坐标以及层级，现在结合本章所学到的知识，是不是可以让《保卫萝卜2》主页面动起来呢？现在我们动手开发吧！



图4-4　《保卫萝卜2》主页面

在开发之前，请原谅我将怪物编号写到图4-4中，虽然这看上去有点丑，但是有助于你更清晰地理解。从《保卫萝卜2》上线作品中得知，1号和5号怪物运行的动作为左右来回运动。2号怪物运行的动作是冲出来之后做上下浮动。3号、4号和6号怪物也是一直上下浮动，6号怪物的手做的是旋转运动，剩下一个萝卜为贝塞尔曲线运动。除了2号怪物的冲出动作和萝卜的动作之外，其他怪物运行的动作都是无限重复执行。

我在MMMainLayer类中定义了一个actionDuration属性作为所有节点动作的时间基数，你可以通过更改此值来加快或放慢整个当前层中所有节点的动作速度，相关代码如下：

```
1  var MMMainLayer = cc.Layer.extend({
2      actionDuration : 1,        // 时间基数[页面上所有节点运行的动作]
3      ctor : function(){
4          this._super();
5            // ......
6      }
7  });
```

然后，以“帮助”按钮为例，在MMMainLayer的loadHelp()函数中追加“帮助”按钮的运行动作，相关代码如下：

```
1  // 加载[帮助]
2  loadHelp : function(){
3      var helpBody = new cc.Sprite("res/MainMenu/front_monster_6.png");
4      // ......
```

```
5       // 上下移动
6       var helpBodyMoveBy1 = cc.moveBy(this.actionDuration * 2, cc.p(0, 5));
7       var helpBodyMoveBy2 = cc.moveBy(this.actionDuration * 2, cc.p(0, -5));
8       var helpBodySeq = cc.sequence(helpBodyMoveBy1, helpBodyMoveBy2);
9       var helpBodyAction = helpBodySeq.repeatForever();
10      helpBody.runAction(helpBodyAction);
11      // ......
12 }
```

在第6行和第7行代码中，我们创建了两个互补的moveBy动作，时间为actionDuration基数的2倍，位移为Y轴方向上上下5个像素。第8行将这两个动作包成一个cc.Sequence类型的序列动作helpBodySeq，第9行通过调用helpBodySeq.repeatForever()函数得到一个无限循环的序列动作helpBodyAction，第10行运行helpBodyAction，便实现了“帮助”按钮一直上下浮动的动作效果。

其他怪物的动作也和“帮助”按钮大同小异，稍有不同的便是2号怪物。2号怪物运行的是一个冲出来之后做上下浮动的动作，此时可以考虑将这个动作分成两个步骤来做，先运行冲出的动作，然后再做上下浮动。那么，此动作也是一个复合动作，代码如下：

```
1  // 加载5号和2号[怪物]
2  loadForeMonster : function(){
3      var leftCambridgeBlue = new cc.Sprite("res/MainMenu/front_monster_2.png");
4      // ......
5      // 上下移动
6      var action0 = cc.moveTo(this.actionDuration * 0.2, cc.p(cc.winSize.width /
         2 - 220, 170));
7      var action1 = cc.sequence(action0, cc.callFunc(function () {
8          var blueMoveBy1 = cc.moveBy(this.actionDuration * 0.55, cc.p(0, -5));
9          var blueMoveBy2 = cc.moveBy(this.actionDuration * 0.55, cc.p(0, 5));
10         var blueSeq = cc.sequence(blueMoveBy1, blueMoveBy2);
11         var blueAction = blueSeq.repeatForever();
12         leftCambridgeBlue.runAction(blueAction);
13     }, this));
14     leftCambridgeBlue.runAction(action1);
15 }
```

除此之外，就剩下萝卜的动作比较特殊一些了，它是一个贝塞尔曲线加缩放并行运动。在运行动作之前，应当将萝卜缩放至原始大小的0.7倍，然后再运行复合动作，相关代码如下：

```
1  // 加载[萝卜]
2  loadCarrot : function(){
3      var node = new cc.Sprite("res/MainMenu/front_carrot.png");
4      // ......
5      // 萝卜，贝塞尔曲线 + 缩放运动
6      node.setScale(0.7);
7      node.setPosition(cc.winSize.width / 2 + 320, 120);
8      var controlPointsTo = [
9          cc.p(cc.winSize.width / 2 + 400, 100),  // 起点控制点
10         cc.p(cc.winSize.width / 2 + 120, 0),    // 终点控制点
11         cc.p(cc.winSize.width / 2 + 100, 20)];  // 终点
12     var bezierTo = cc.bezierTo(this.actionDuration * 0.8, controlPointsTo);
```

```
13     var scaleTo = cc.scaleTo(this.actionDuration * 0.8, 1);
14     var spawn = cc.spawn(bezierTo, scaleTo);
15     node.runAction(spawn);
16 }
```

其余动作都像“帮助”按钮的动作那样简单，也都是差不多重复的代码，详情可参看随书代码。

4.6 小结

通过本章的学习，我们知道了在Cocos2d-JS中，大部分的动作可分为瞬时动作和持续动作，瞬时动作在一帧内完成，而持续动作则会持续一段时间。除此之外，Cocos2d-JS提供了复合和变速等动作，你可以把简单的动作通过复合动作再构成一个复杂的动作，并且复合动作`cc.Sequence`将一组普通动作包装成一个序列，然后按照顺序挨个执行。而`cc.Spawn`是同时并发执行所有包含在`cc.Spawn`里面的动作，不需要注意动作放置进去的先后顺序。最后，在本章的实例中，我们让《保卫萝卜2》的主页面动了起来。

4.7 参考资源

本章的参考资源为Cocos2d-x官方文档（动作篇）：http://www.cocos.com/doc/article/index?type=cocos2d-x&url =/doc/cocos-docs-master/manual/framework/native/v3/action/zh.md。

第 5 章

事件机制

随着软件和硬件的迅速发展，游戏所能达到的效果也越来越接近电影了，游戏和电影一样，都可以有动人的故事情节以及炫丽的动作特效等。但是，游戏和电影又有着很大的区别，其中最大的区别就是游戏需要和用户交互。用户交互是游戏的根本，甚至可以说，没有用户交互，就没有游戏。用户交互可以是触摸屏幕、重力感应、键盘响应和鼠标响应等，而这些在游戏中都属于事件。

本章内容：

- 事件
- 事件管理器
- 触摸事件
- 加速计事件
- 键盘事件
- 鼠标事件
- 自定义事件
- 实例——虚拟摇杆封装

5.1 事件

Cocos2d-JS为处理事件封装了3个相关的类，每个事件的处理都是由这三个部分组成，它们分别是`cc.Event`（事件对象）、`cc.EventListener`（事件监听器）和`cc.EventManager`（事件管理器）。`cc.Event`对象携带着事件相关的信息，例如事件类型、事件回调函数等。`cc.EventListener`封装了用户的事件处理逻辑，而`cc.EventManager`则管理用户注册的事件监听器，根据触发的事件类型分发给相应的事件监听器。

每个事件的事件源可以是层、精灵、菜单等节点对象，事件的逻辑处理都在各个事件监听器（`cc.EventListener`）中。每个事件监听器都需要被添加到事件管理器中，这使得当有事件被触发时，事件管理器能够根据事件的类型把事件分发给相应的事件监听器，事件监听器收到事件之后，开始处理事件的业务逻辑。

Cocos2d-JS提供了5种事件监听器，它们的继承关系如图5-1所示。

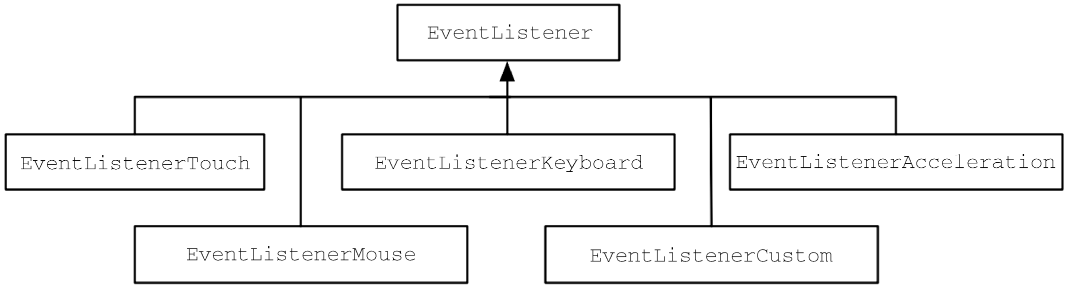


图5-1　事件监听器的继承关系

说明　在Cocos2d-x中，有事件分发器的概念。实际上，Cocos2d-JS中的事件管理器就是事件分发器。我们可以在frameworks/js-bindings/bindings/script/jsb_boot.js中看到这样一行代码：

```
1 cc.eventManager = cc.director.getEventDispatcher();
```

5.2　事件管理器

顾名思义，事件管理器就是负责管理事件监听器的，它是事件监听器的大总管，管理事件监听器的添加、删除以及分发。`cc.EventManager`是一个单例类，`cc.eventManager`为`cc.EventManager`的单例对象，它通过下面这些函数管理各类事件监听器。

5.2.1　添加事件监听器

每个事件监听器都需要添加到事件管理器对象中，添加事件监听器的函数如下：

```
1  cc.eventManager.addListener(listener, nodeOrPriority);
```

其中，第一个参数`listener`是事件监听器对象，第二个参数`nodeOrPriority`是一个Node对象或者是一个数值。若给定的参数为数值，则数值越低，事件的优先级越高。若给定的参数为一个Node对象，则事件优先级取决于Node对象的`LocalZOrder`，该值越大，事件优先级越高。

除此之外，`cc.EventListener`还允许添加自定义事件。自定义事件取代了Cocos2d-x 2.x版本中的`CCNotificationCenter`。你可以通过如下API添加自定义事件：

```
1  cc.eventManager.addCustomListener(eventName, callback);
```

通常情况下，`eventName`为自定义事件的名字，是字符串类型，`callback`为事件的回调函数。

5.2.2　删除事件监听器

删除和添加总是成对存在的，当你不再需要事件监听器的时候，可以通过cc.EventListener的如下函数移除：

```
1  // 删除指定监听器
2  cc.eventManager.removeListener(listener);
3  // 删除指定类型监听器
4  cc.eventManager.removeListeners(listenerType, recursive);
5  // 删除指定自定义监听器
6  cc.eventManager.removeCustomListeners(customEventName);
7  // 删除所有监听器
8  cc.eventManager.removeAllListeners();
9  // 删除target下指定类型的监听器
10 cc.eventManager.removeEventListener(type, target);
```

5.2.3　设置事件监听器的优先级

每个事件监听器都具有优先级，优先级可以通过如下代码给事件设置或更改：

```
1  cc.eventManager.setPriority(listener, fixedPriority);
```

另外，前面提到过，在cc.eventManager中添加事件监听器的时候，便可直接设定事件监听器的优先级，示例代码如下：

```
1  cc.eventManager.addListener(listenerA, -1000);
2  cc.eventManager.addListener(listenerB, -500);
```

将代码转为示意图，就如图5-2所示，Node_A和Node_B都注册了触摸事件，Node_A的触摸事件优先级是−1000，Node_B的触摸优先级是−500，虽然Node_B在Node_A的上方，但是Node_A却优先被触摸到。

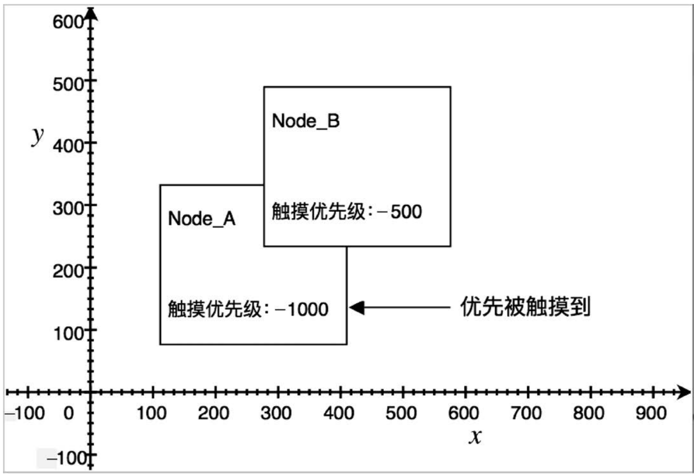


图5-2　触摸优先级

或许你已经注意到了，此时，你并没有看到事件监听器是如何和需要监听的节点进行关联的。实际上，每个事件监听器都有target属性，它指向需要被监听的事件对象，也就是事件源，具体代码见5.3.1节。

5.2.4 分发事件

当有事件被触发时，cc.eventManager会将收到的事件分发给对应的事件监听器。事件分发的相关API代码如下：

```
1  cc.eventManager.dispatchEvent(event);        // 分发事件
2  // 分发自定义事件
3  cc.eventManager.dispatchCustomEvent(eventName, userData);
```

第1行代码为cc.eventManager分发Cocos2d-JS封装好的事件，参数event是一个事件对象。而第3行代码中的eventName参数为用户自定义事件的名字，是一个字符串类型，userData为事件携带的数据，可以是一个对象等。另外，userData参数可省略。实际上，cc.eventManager.dispatchCustomEvent(eventName,userData)函数是对cc.eventManager.dispatchEvent-(event)函数的二次封装，它的源码如下：

```
1  cc.eventManager.dispatchCustomEvent = function(eventName, userData){
2      var ev = new cc.EventCustom(eventName);
3      ev.setUserData(optionalUserData);
4      this.dispatchEvent(ev);
5  };
```

5.3 触摸事件

毫无疑问，在移动设备上，触摸事件是用户交互方式中最为基础的一种，通常包含点击、滑动、抬起。触摸又分单点触摸和多点触摸两种。

5.3.1 单点触摸

Cocos2d-JS将单点触摸标记为cc.EventListener.TOUCH_ONE_BY_ONE，你可以通过如下代码创建一个单点触摸的事件监听器：

```
1  var listener = cc.EventListener.create({
2      event : cc.EventListener.TOUCH_ONE_BY_ONE,
3      swallowTouches : true,    // 是否吞噬触摸
4      target : this,  // 指定事件源
5      onTouchBegan: function (touch, event) {
6          // 获取[当前触发事件的对象]
7          var target = event.getCurrentTarget();
8          // 将点击坐标转换为基于当前触发事件对象的本地坐标
9          var posInNode = target.convertToNodeSpace(touch.getLocation());
10         // 获取当前节点大小
```

```
11          var size = target.getContentSize();
12          // 区域设定
13          var rect = cc.rect(0, 0, size.width, size.height);
14          // 判断触摸点是否在节点区域内
15          if (!(cc.rectContainsPoint(rect, posInNode))) {
16              return false;
17          }
18          // TODO something
19          return true;
20      },
21      onTouchMoved: function (touch, event) {
22          var target = event.getCurrentTarget();
23          var delta = touch.getDelta(); // 获取[滑动距离]
24          target.x += delta.x;
25          target.y += delta.y;
26      },
27      onTouchEnded: function (touch, event) {
28          // TODO something
29      },
30 });
```

第1行代码通过`cc.EventListener`类创建了一个事件监听器对象,第2行代码指定当前监听器监听的事件类型为`TOUCH_ONE_BY_ONE`，表示单点触摸，而第3行的`swallowTouches`表示是否吞噬触摸事件，当`swallowTouches`被设置为`true`之后，触摸事件将不再向下传递。

假设你现在有两个事件监听器`listenerA`和`listenerB`，`listenerA`的事件优先级比`listenerB`高，`listenerA`监听节点Node_A，`listenerB`监听节点Node_B。若`listenerA`中事件对象的`swallowTouches`值为`true`的话，那么点击Node_A触发`listenerA`时，触摸事件被吞噬，`listenerB`将不被触发，即Node_B不会接收到触摸事件，如图5-3所示。若`listenerA`的`swallowTouches`值为`false`，则`listenerA`和`listenerB`将依次被触发，如5-4所示。

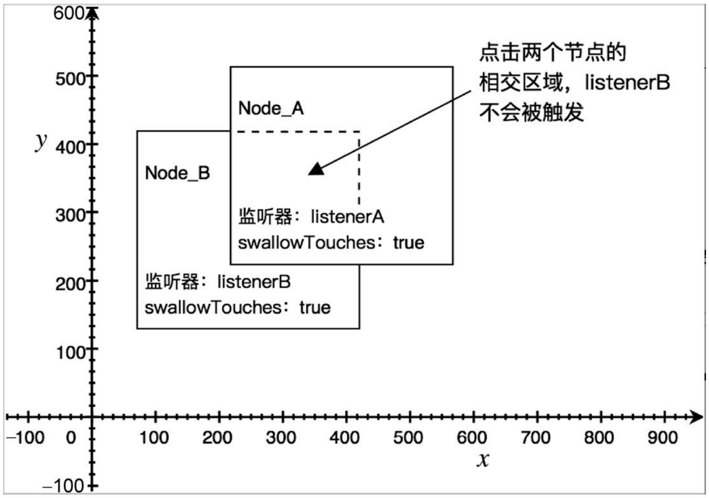


图5-3　`swallowTouches`为`true`的事件吞噬情况

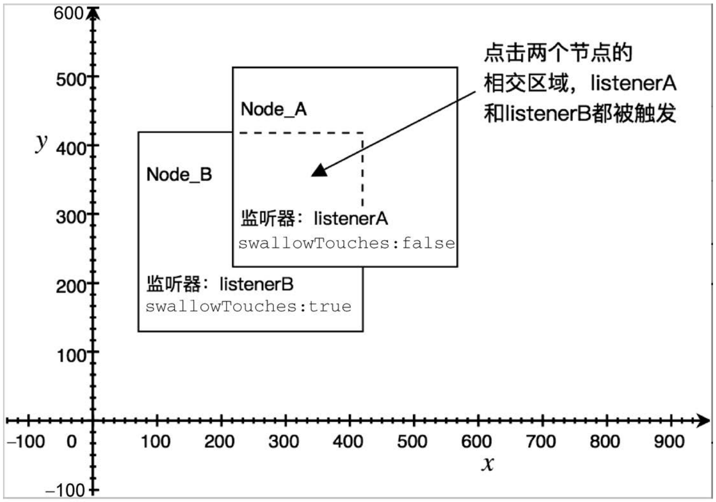


图5-4 swallowTouches为false的事件吞噬情况

第5行的onTouchBegan、第21行的onTouchMoved和第27行的onTouchEnded为触摸事件的回调函数，分别表示触摸开始、触摸移动、触摸结束。值得注意的是，在这三个方法当中，this并不是指向当前监听器监听的节点，而是指向事件监听器本身。所以，通常所要获取的this对象实际上是事件监听器的target（监听对象，事件源），target一般是cc.Node或者cc.Node的子类，例如cc.Layer、cc.Sprite等。通过在第7行代码，可以获取到这个target。

当获取到target之后，需要判断当前触摸点是否正确触摸到了target，第8行到第17行就是做了这一件事情。首先，将触摸点转为target的本地坐标，然后获取下target的大小，再根据target的大小生成一块区域，再通过cc.rectContainsPoint(rect, posInNode)函数判断转换后的posInNode是否在target的区域内，若不是，则返回false，若是，继续执行代码，最后返回true。当返回false之后，onTouchMoved和onTouchEnd将不被调用。

第23行是一个非常有用的功能，它返回上一次onTouchMoved和这一次onTouchMoved的触摸偏移量，并且有正负值，利用这个值，可以实现移动target，也可以实现滑动方向的获取，这在平常的开发中，是非常有用的一个功能。

当事件监听器创建完毕之后，通过下面代码便可以对指定的节点添加触摸事件监听：

```
1  cc.eventManager.addListener(listener, sprite1);
2  cc.eventManager.addListener(listener.clone(), sprite2);
3  cc.eventManager.addListener(listener.clone(), sprite3);
```

5.3.2 多点触摸

多点触摸被标记为cc.EventListener.TOUCH_ALL_AT_ONCE。与单点触摸不同的是，多点触摸不支持事件吞噬，也就是说，多点触摸没办法穿透下去。另外，多点触摸的事件相关处理

函数是onTouchesBegan、onTouchesMoved、onTouchesEnded，这与单点触摸也有所不同，如下面代码中的第5行到第7行所示：

```
1  // 判断当前平台是否支持多点触摸
2  if( 'touches' in cc.sys.capabilities ) {
3      var listener = cc.EventListener.create({
4          event            : cc.EventListener.TOUCH_ALL_AT_ONCE,
5          onTouchesBegan   : this.onTouchesBegan,
6          onTouchesMoved   : this.onTouchesMoved,
7          onTouchesEnded   : this.onTouchesEnded
8      });
9      cc.eventManager.addListener(listener, this);
10 }else{
11     // 当前平台不支持多点触摸
12 }
```

当然，在例如onTouchesBegan、onTouchesMoved和onTouchesEnded事件处理函数中，获取出多个触摸点，代码如下：

```
1  onTouchesBegan: function (touches, event) {
2      var self = event.getCurrentTarget();
3      // touches[0]表示捕获第一个触摸点，touches为触摸点集合
4      for (var i = 0; i < touches.length;i++ ) {
5          var touch = touches[i];
6          var pos = touch.getLocation();
7          var id = touch.getID();
8          // 获取点击坐标[基于本地坐标]
9          var locationInNode = self.convertToNodeSpace(touch.getLocation());
10         // 获取当前节点大小
11         var size = self.getContentSize();
12         // 区域设定
13         var rect = cc.rect(0, 0, size.width, size.height);
14         // 判断触摸点是否在节点区域内
15         if (!(cc.rectContainsPoint(rect, locationInNode))) {
16             return false;
17         }
18         cc.log("onTouchesBegan at: " + pos.x + " " + pos.y + " Id:" + id);
19         return true;
20     }
21 }
```

5.4　加速计事件

或许对不熟悉移动应用开发的朋友来说，加速计这个词可能有点陌生，因为我们更常听到的是另外一个名词——重力感应。其实，重力感应这个说法并不准确，因为加速计感应的并非只有一个重力加速度（g），还包括设备本身移动的加速度。当只有设备被静止放置在地上的时候，加速计所接收到的所有加速度的矢量和才会等于重力加速度，这时候才算是重力感应。

加速计会感应到x、y和z三个方向上的加速度分量，x值是水平方向上的分量，y值是竖直方向上的分量，而z值是垂直于屏幕的方向上的分量。另外，加速计感应事件和前面提到的触摸事件有所不同，加速计事件需要cc.inputManager对象开启加速计之后才可以正常使用。另外，

加速计事件的回调函数会接收到一个携带加速度分量数值的参数，代码如下：

```
1  // 判断当前平台是否支持加速计
2  if( 'accelerometer' in cc.sys.capabilities ) {
3      // 开启加速计
4      cc.inputManager.setAccelerometerEnabled(true);
5      var listener = cc.EventListener.create({
6          event      : cc.EventListener.ACCELERATION,
7          callback : function (acc, event) {
8              cc.log("accX = ", acc.x);       // x分量
9              cc.log("accY = ", acc.y);       // y分量
10             cc.log("accZ = ", acc.z);       // z分量
11         }
12     });
13     cc.eventManager.addListener(listener, this);
14 }else{
15     cc.log("不支持加速计事件");
16 }
```

除此之外，你可以通过cc.inputManager.setAccelerometerInterval(number)函数来设置加速计监听的间隔时间，其默认值为1/30，一般保持默认值即可。

5.5 键盘事件

移动设备虽然已经偏向屏幕触摸操作多年，但是设备上还是存在一些非虚拟按键（物理按键或者触感按键），加上Cocos2d-JS是跨全平台的游戏引擎，支持发布到Mac OS X以及Windows等平台上，所以，Cocos2d-JS将这些按键事件都统一封装为键盘事件。按键事件的回调函数会接收到一个携带按键编码（KeyCode）的参数，例如：

```
1  // 判断当前平台是否支持键盘事件
2  if( 'keyboard' in cc.sys.capabilities ) {
3      var listener = cc.EventListener.create({
4          event          : cc.EventListener.KEYBOARD,
5          target         : this,
6          onKeyPressed   : function(keyCode, event){
7              var keyCodeStr = String.fromCharCode(keyCode); //按键编码转为字符
8              cc.log(keyCodeStr + "(" + keyCode.toString() + ")按下");
9          },
10         onKeyReleased  : function(keyCode, event){
11             var keyCodeStr = String.fromCharCode(keyCode);//按键编码转为字符
12             cc.log(keyCodeStr + "(" + keyCode.toString() + ")释放");
13         }
14     });
15     cc.eventManager.addListener(listener, this);
16 }else{
17     cc.log("键盘事件不支持");
18 }
```

可以看出，键盘事件的回调函数有两个，onKeyPressed函数在按键被按下时触发，onKeyReleased在按键被松开时触发，一般可以考虑在onKeyReleased函数中做逻辑处理。运

行上面代码，然后任意按下一个键（我按下S键），可以在控制台看到如图5-5所示的输出结果。

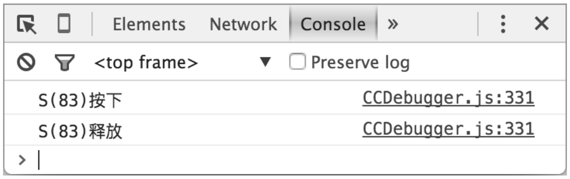


图5-5　键盘事件测试控制台输出

5.6　鼠标事件

虽然触摸事件也支持电脑上鼠标的点击、拖曳、抬起事件，但是触摸事件并不支持鼠标右键按下、滚轮滚动等操作，这是因为它们确确实实属于鼠标事件。Cocos2d-JS鼠标事件标记为cc.EventListener.MOUSE，提供4个鼠标事件回调函数，分别是：按键按下（onMouseDown）、按键松开（onMouseUp）、鼠标移动（onMouseMove）和滚轮滚动（onMouseScroll）。鼠标事件的使用代码如下：

```
1  // 判断当前平台是否支持鼠标事件
2  if( "mouse" in cc.sys.capabilities ) {
3      var listener = cc.EventListener.create({
4          event       : cc.EventListener.MOUSE,
5          target      : this,
6          onMouseDown : function(event){      // 鼠标事件[按下]
7              var pos = event.getLocation();
8              var button = event.getButton();
9              if (button == cc.EventMouse.BUTTON_LEFT){        // 左键
10                 cc.log("左键按下", + pos.x + " " + pos.y);
11             }else if(button == cc.EventMouse.BUTTON_RIGHT){ // 右键
12                 cc.log("右键按下", + pos.x + " " + pos.y);
13             }else if(button == cc.EventMouse.BUTTON_MIDDLE){// 滚轮
14                 cc.log("中间滚轮键按下");
15             }
16         },
17         onMouseUp   : function(event){      // 鼠标事件[抬起]
18         },
19         onMouseMove: function(event){       // 鼠标事件[移动]
20             var pos = event.getLocation();
21             cc.log("鼠标当前位置：", pos.x, pos.y)
22         },
23         onMouseScroll:function(event){      // 鼠标事件[滚轮滚动]
24             var pos = cc.p(event.getScrollX(), event.getScrollY());
25             cc.log("鼠标滚轮滚动x ：", pos.x + " y: " + pos.y);
26         }
27     });
28     cc.eventManager.addListener(listener, this);
29 }else{
30     cc.log("不支持鼠标事件");
31 }
```

运行上面代码，然后移动鼠标、点击鼠标的左右键、滑动滚轮，可以在控制台看到如图5-6所示的输出信息，其中滚动轮输出值的正负表示滚动方向。

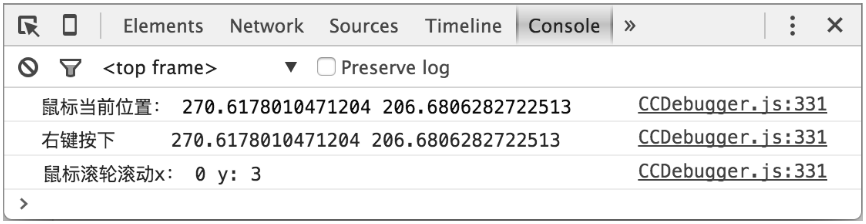


图5-6 鼠标事件测试控制台输出

5.7 自定义事件

除了上述的几个事件之外，引擎还封装了一个自定义事件，它取代了Cocos2d-x 2.x版本中的通知中心CCNotificationCenter。下列代码创建了一个自定义事件，并将其添加到事件管理器中：

```
1  // 创建自定义函数
2  var listener = cc.EventListener.create({
3      event       : cc.EventListener.CUSTOM,
4      target      : this,              // 事件源
5      eventName   : "custom_event",    // 事件名
6      callback    : function(event){   // 回调函数
7          var target = event.getCurrentTarget();  // 事件源对象
8          cc.log("自定义事件被触发...\n携带数据：", event.getUserData());
9      }
10 });
11 cc.eventManager.addListener(listener, this);
```

上面第5行代码指定了该自定义事件的事件名称，其类型是字符串类型，该属性用于区分自定义事件。在实际开发中，你应该将自定义事件名定义成一个常量，例如cc.EventListener.TOUCH_ONE_BY_ONE。

分发自定义事件需要先根据事件名创建出指定的自定义事件对象，你可以给创建出来的对象设置一个用户数据，这样便可以在事件分发的时候将数据传输给事件监听者，然后通过事件管理器分发自定义事件，代码如下：

```
1  var event = new cc.EventCustom("custom_event"); // 创建自定义事件
2  var data = {          // 定义数据
3      name : "凌建风",
4      age : 23,
5      hobby : "吃饭、睡觉、写代码！"
6  };
7  event.setUserData(data);
8  cc.eventManager.dispatchEvent(event);    // 分发事件
```

分发事件之后，在浏览器的控制台上可以看到如图5-7所示的输出：



图5-7　自定义事件测试控制台的输出

5.8　实例——虚拟摇杆封装

在一些2.5D游戏中，你可能需要一个虚拟摇杆（如图5-8所示）来控制人物的8个方向行走，Cocos2d-JS本身没有封装好的虚拟摇杆（下面简称摇杆），那么，在本章中我们一起来开发一个吧。

我想你应该知道，摇杆需要两张图片，分别是底图和把手，如图5-9和图5-10所示。



图5-8　虚拟摇杆



图5-9　虚拟摇杆底图

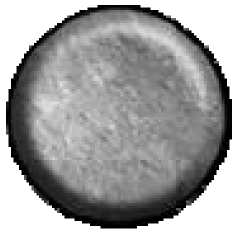


图5-10　虚拟摇杆把手

从图5-8可以看到，摇杆把手放在摇杆底图的正中心。由于精灵的原点也是在正中心，所以将要封装的摇杆继承自精灵类（cc.Sprite）是最合适不过的了。我把摇杆的代码编写在随书代码unit05_event/Rocker.js文件中，下面开始编码。

首先，我给摇杆定义几个类型，分别是默认类型（DEFAULT）、自动类型（AUTO）、隐藏类型（HIDE）以及不透明类型（OPACITY），代码如下：

```
1  // 虚拟摇杆类型
2  var ROCKER_TYPE = ROCKER_TYPE || {};
3  ROCKER_TYPE.DEFAULT = "DEFAULT";// 默认类型
4  ROCKER_TYPE.AUTO = "AUTO";      // 自动类型
5  ROCKER_TYPE.HIDE = "HIDE";      // 隐藏类型
6  ROCKER_TYPE.OPACITY = 255;      // 不透明类型
```

其中，默认类型的位置是固定的，而自动类型则是基于默认类型，加了每次用户点击哪里摇杆就出现在哪里的功能。隐藏类型和不透明类型都是基于自动类型的，只是前者在游戏中不可见，

后者可以设置透明度。虽然隐藏类型可以用不透明类型模拟，但是，在第3章中我已经说过，当一个节点的不透明值为0的时候，渲染器依旧会对其渲染，造成性能浪费，所以，你应当明白节点的不透明值为0和Visible属性值为false的区别。

接着，我定义了ROCKER_DIRECTION对象来保存摇杆的8个方向和原点，把它们都放到了ROCKER_DIRECTION_ARRAY数组中，形成了一个方向和数组下标的映射关系，并且定义了每个方向的角度数组。图解如图5-11所示，相关代码如下：

```
1  // 摇杆方向
2  var ROCKER_DIRECTION = ROCKER_DIRECTION || {};
3  ROCKER_DIRECTION.RIGHT          = "RIGHT";      // 向右
4  ROCKER_DIRECTION.RIGHT_UP       = "RIGHT_UP";   // 右上
5  ROCKER_DIRECTION.UP             = "UP";         // 向上
6  ROCKER_DIRECTION.LEFT_UP        = "LEFT_UP";    // 左上
7  ROCKER_DIRECTION.LEFT           = "LEFT";       // 向左
8  ROCKER_DIRECTION.LEFT_DOWN      = "LEFT_DOWN";  // 左下
9  ROCKER_DIRECTION.DOWN           = "DOWN";       // 向下
10 ROCKER_DIRECTION.RIGHT_DOWN     = "RIGHT_DOWN"; // 右下
11 ROCKER_DIRECTION.ORIGIN         = "ORIGIN";     // 原点
12 // 方向数组
13 var ROCKER_DIRECTION_ARRAY = [
14     ROCKER_DIRECTION.RIGHT,
15     ROCKER_DIRECTION.RIGHT_UP,
16     ROCKER_DIRECTION.UP,
17     ROCKER_DIRECTION.LEFT_UP,
18     ROCKER_DIRECTION.LEFT,
19     ROCKER_DIRECTION.LEFT_DOWN,
20     ROCKER_DIRECTION.DOWN,
21     ROCKER_DIRECTION.RIGHT_DOWN,
22     ROCKER_DIRECTION.ORIGIN
23 ];
24 // 8个方向的角度数组
25 var ROCKER_ANGLE_ARRAY = [22.5,67.5,112.5,157.5,202.5,247.5,292.5,337.5];
```

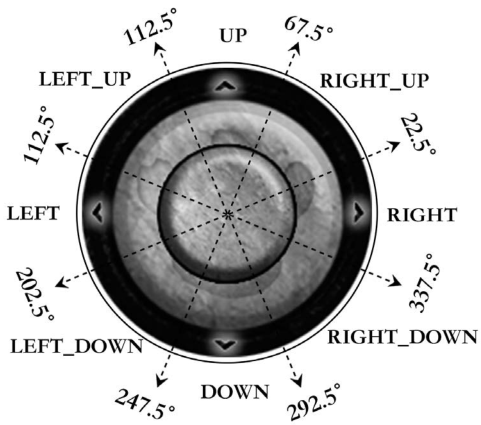


图5-11 虚拟摇杆方向的角度映射关系

接着我定义了摇杆类Rocker，它继承自cc.Sprite，代码如下：

```
1  // 摇杆精灵
2  var Rocker = cc.Sprite.extend({ // 继承cc.Sprite类
3      ctor : function(baseTexture, knobTexture, type){
4          this._super(); // 调用父类构造函数
5          return true;
6      }
7  });
```

在Rocker的构造函数中，接收了3个参数，分别是摇杆底图资源、摇杆把手资源以及摇杆类型。毫无疑问，摇杆底图和摇杆把手都是一个精灵，并且它们在Rocker类的其他方法里会访问到，所以，应该把它们定义成Rocker类的属性，摇杆的类型也不例外。此外，还需要定义成类成员属性的还有触摸监听事件touchListener、把手可移动半径radius以及方向direction等。具体代码如下：

```
1  var Rocker = cc.Sprite.extend({
2      _baseNode        : null,      // 底盘[节点]
3      _knobNode        : null,      // 把手[节点]
4      _touchListener   : null,      // 触摸事件[监听器]
5      _radius          : 0,         // 摇杆的可移动半径
6      _angle           : 0,         // 角度
7      _velocity        : cc.p(0, 0),// 速度
8      _callback        : null,      // 回调函数
9      _direction       : ROCKER_DIRECTION.ORIGIN, // 方向
10     _type            : ROCKER_TYPE.DEFAULT,     // 摇杆类型
11     ctor: function(baseTexture, knobTexture, type){
12         this._super();
13         return true;
14     }
15 });
```

其中，velocity为摇杆的速度向量，取值为[0, 1]。_callback为摇杆的回调函数，我把它声明为私有属性，然后通过cc.defineGetterSetter(proto, prop, getter, setter)函数为_callback定义了属性操作，代码如下：

```
1  var _p = Rocker.prototype;
2  cc.defineGetterSetter(_p, "callback", _p.getCallback, _p.setCallback);
3  _p.callback;
```

```
1  getCallback : function(){
2      return this._callback;
3  },
4  setCallback : function(callback){
5      this._callback = callback;
6  }
```

下面定义两个精灵来表示摇杆的底图和把手，代码如下：

```
1  // 加载精灵[_baseNode和_knobNode]
2  loadBaseAndKnob : function(baseTexture, knobTexture){
3      this._baseNode = new cc.Sprite(baseTexture);
4      this._knobNode = new cc.Sprite(knobTexture);
```

```
5      this.addChild(this._baseNode);
6      this.addChild(this._knobNode);
7  }
```

再编写一个函数，用来配置半径radius和类型type等属性，代码如下：

```
1  // 加载配置半径_radius和类型type等
2  loadConfig : function(type){
3      this._radius = this._baseNode.getContentSize().width / 2;
4      if (type !== undefined){
5          if (isNaN(type)){
6              this._type = type;
7              if (this._type == ROCKER_TYPE.HIDE){
8                  this.setVisible(false);
9              }
10         }else{
11             this._type = ROCKER_TYPE.OPACITY;
12             this.setCascadeOpacityEnabled(true); // 开启子节点透明度关联
13             this.setOpacity(type);
14         }
15     }
16 }
```

此时，在Rocker的构造函数中，分别调用loadBaseAndKnob(baseTexture, knobTexture)和loadConfig(type)函数。

接下来，开始对摇杆添加（注册）触摸事件。这里我编写了两个和触摸事件相关的函数，分别是注册事件registerEvent和卸载事件unRegisterEvent，代码如下：

```
1  // 注册[触摸事件监听器]
2  registerEvent : function(){
3      this._touchListener = cc.EventListener.create({
4          event           : cc.EventListener.TOUCH_ONE_BY_ONE,
5          target          : this,
6          swallowTouches  : true,
7          onTouchBegan    : this.onTouchBegan,
8          onTouchMoved    : this.onTouchMoved,
9          onTouchEnded    : this.onTouchEnded
10     });
11     cc.eventManager.addListener(this._touchListener, this);
12 },
13 // 卸载[触摸事件监听器]
14 unRegisterEvent : function(){
15     cc.eventManager.removeListener(this._touchListener);
16 }
```

整个摇杆中，比较重要的部分就是触摸事件回调函数中的逻辑了。在onTouchBegan函数中，主要是根据不同类型的摇杆做对应的处理，例如默认类型的摇杆，则限定必须触摸到把手摇杆才能正确工作，并且整个摇杆的位置不会发生变动。如若不是默认类型的摇杆，则将整个摇杆的坐标设置为当前触摸点。除此之外，如果是自动类型的，则将整个摇杆显示出来。代码如下：

```
1  onTouchBegan: function (touch, event) {
2      var target = this.target;
```

```
3       var knob = target._knobNode; // 获取把手
4       var locationInNode = knob.convertToNodeSpace(touch.getLocation());
5       var size = knob.getContentSize();
6       var rect = cc.rect(0, 0, size.width, size.height);
7       if (target._type == ROCKER_TYPE.DEFAULT){ // 默认类型
8           if (!cc.rectContainsPoint(rect, locationInNode)) {
9               return false;
10          }
11      }else { // 非默认类型
12          if (target._type == ROCKER_TYPE.AUTO){
13              target.setVisible(true);
14          }
15          target.setPosition(touch.getLocation());
16      }
17      return true;
18  }
```

在onTouchMoved函数中，主要就是计算出摇杆的角度、方向以及速度等，另外还限制了把手和原点的距离不能超过摇杆的半径，代码如下：

```
1   onTouchMoved: function (touch, event) {
2       // 节点获取
3       var target = this.target;
4       var knob = target._knobNode;
5       // 触摸点转为摇杆的本地坐标
6       var locationInNode = target.convertToNodeSpace(touch.getLocation());
7       target.onUpdate(locationInNode); // 角度、方向、速度等更新
8       // 长度获取[当前触摸点相对摇杆中心点]
9       var length = cc.pLength(locationInNode);
10      var radians = cc.degreesToRadians(target._angle);
11      // 限制把手和原点的距离不能超过摇杆的半径
12      if ( length > target._radius){
13          var x = Math.cos(radians) * target._radius;
14          var y = Math.sin(radians) * target._radius;
15          knob.setPosition(cc.p(x, y));
16      }else{
17          knob.setPosition(locationInNode);
18      }
19  }
```

上面代码中，第7行是一个总更新函数，此函数也只在onTouchMoved中被调用，用来更新摇杆的角度、方向以及速度，代码如下：

```
1   // 更新[角度、方向、速度]
2   onUpdate : function(pos){
3       // 更新[角度]
4       this.onUpdateAngle(pos);
5       // 更新[方向]
6       this.onUpdateDirection(pos);
7       // 更新[速度]
8       this.onUpdateVelocity();
9   }
```

从上面的代码中可以看到，onUpdate函数分别调用了更新角度、方向、速度的函数。在

onUpdateAngle函数中，接收一个点，通过cc.pToAngle(pos)函数获得此点pos和原点cc.p(0, 0)所形成的线段与X轴正方向夹角的弧度，然后通过cc.radiansToDegrees(radians)函数将弧度转为角度，并且将角度转为[0, 360]的区间，而不是–112.5°这样的值。转为360°的方式，是与***ROCKER_ANGLE_ARRAY***数组配合工作，具体下面会讲到。onUpdateAngle的代码如下：

```
1  // 更新[角度]
2  onUpdateAngle : function(pos){
3      this._angle = cc.radiansToDegrees(cc.pToAngle(pos));
4      if (this._angle < 0) {
5          this._angle += 360;
6      }
7  }
```


说明 cocos2d-x/web/cocos2d/core/support/CCPointExtension.js文件中定义了许多关于点或向量的常用函数，例如cc.pDistance(v1, v2)可以返回两点之间的距离等。

当角度转为[0, 360]的取值区间，再加上方向数组ROCKER_DIRECTION_ARRAY以及8个方向的角度数组ROCKER_ANGLE_ARRAY的辅助，获取方向就变得比较简单了。只需遍历一下ROCKER_ANGLE_ARRAY数组，然后将当前的角度与ROCKER_ANGLE_ARRAY中的值进行区间匹配，匹配成功后得到索引。因为ROCKER_DIRECTION_ARRAY和ROCKER_ANGLE_ARRAY这两个数组存在映射关系，所以可以直接通过当前的索引取出对应的方向。更新方向的代码如下：

```
1  // 更新[方向]
2  onUpdateDirection : function(){
3      for (var i = 1; i < ROCKER_ANGLE_ARRAY.length; i++) {
4          this._direction = ROCKER_DIRECTION_ARRAY[0]; // 默认向右
5          if (this._angle >= ROCKER_ANGLE_ARRAY[i - 1] && this._angle < ROCKER_ANGLE_
               ARRAY[i]) {
6              this._direction = ROCKER_DIRECTION_ARRAY[i];
7              break;
8          }
9      }
10 }
```

前面说到，我将速度设置成一个向量，其X轴和Y轴的分量的取值范围都为[0, 1]。速度更新代码如下：

```
1  // 更新[速度]
2  onUpdateVelocity : function(){
3      this._velocity.x = this._knobNode.getPositionX() / this._radius;
4      this._velocity.y = this._knobNode.getPositionY() / this._radius;
5  }
```

而在onTouchEnded函数中做了一些收尾的工作。例如，如果当前的Rocker类型是自动的，则将摇杆隐藏，并且对摇杆进行重置操作。此外，当速度等被重置后，摇杆把手回到原点后也触发了回调函数。该函数的代码如下：

```
1  onTouchEnded: function (touch, event) {
2      var target = this.target;
3      if (target._type == ROCKER_TYPE.AUTO){
4          target.setVisible(false);
5      }
6      target.reset();          // 重置
7      target.onCallback();     // 触发回调函数
8  },
9  // 重置
10 reset : function(){
11     this._knobNode.setPosition(0, 0);
12     this._angle = 0;
13     this._velocity = cc.p(0, 0);
14     this._direction = ROCKER_DIRECTION.ORIGIN;
15 },
16 // 触发[回调函数]
17 onCallback : function(){
18     (this._callback && typeof(this._callback) === "function") && this._callback
          (this._velocity);
19 }
```

最后，还需要启动update调度器，用来触发回调函数。该函数的实现代码如下：

```
1  update : function(dt){
2      if (this._direction != ROCKER_DIRECTION.ORIGIN) {
3          this.onCallback();  // 触发回调函数
4      }
5  }
```

至此，摇杆封装完毕，可以通过如下代码（完整代码见随书代码）创建出一个摇杆：

```
1  // 加载[摇杆]
2  loadRocker : function(){
3      var node = new Rocker(res.u5_control_base_png, res.u5_control_knob_png,
           ROCKER_TYPE.DEFAULT);
4      this.addChild(node);
5      node.setCallback(function(vec){
6          cc.log("---------------");
7          cc.log("速度，x：", vec.x, " y：", vec.y);
8          cc.log("角度：", node.angle);
9          cc.log("方向：", node.direction);
10     }.bind(this));
11     node.setPosition(200, 130);
12 }
```

其中，Rocker构造函数中的最后一个参数ROCKER_TYPE.DEFAULT也可以是ROCKER_TYPE.AUTO、ROCKER_TYPE.HIDE或者是一个0到255的数值，它表示摇杆的类型。最后，运行代码，移动摇杆，可以在控制台看到如图5-12所示的输出结果。当触摸结束，把手回到原点，速度、角度都为0了，方向重置为ORIGIN。

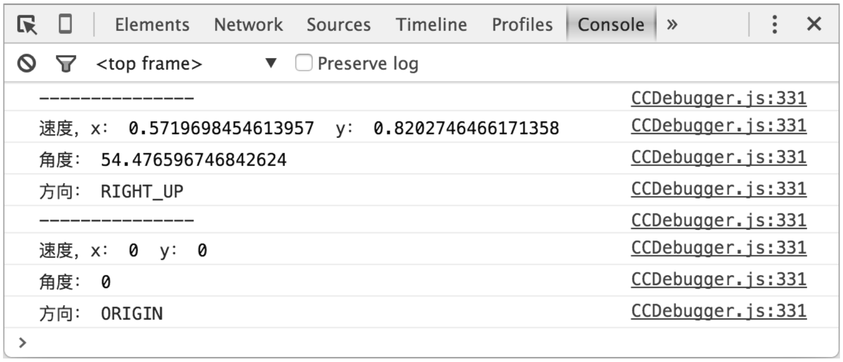


图5-12　虚拟摇杆测试控制台的输出结果

5.9　小结

通过本章的学习，我们知道了在Cocos2d-JS中，每个事件的处理都由事件对象、事件监听器、事件管理器互相配合而成。事件对象中保存着事件的类型、相关回调函数等信息。Cocos2d-JS封装好了触摸事件、加速计感应事件、键盘事件、鼠标事件以及自定义事件，自定义事件取代了Cocos2d-x 2.x版本的通知中心`CCNotificationCenter`，我们应该学会并合理使用自定义事件，这对代码的低耦合有很大的帮助。另外，在本章的实例中，我们一起封装了一个虚拟摇杆。

5.10　参考资源

本章的参考资源为事件分发机制：http://www.cocos2d-x.org/docs/manual/framework/html5/v3/eventManager/zh。

第 6 章

音频处理

一款成功的游戏，不仅需要美术视觉上的盛宴，更需要音频处理带来的听觉冲击。音频不仅可以调动游戏的背景气氛，控制游戏节奏，增强游戏代入感，还可以在场景切换加载资源的时候起到“人未到声先到”的过渡作用，缓解玩家等待资源加载的烦躁。

在游戏中，音频通常分为音乐和音效两种。音乐即为游戏背景音乐，用来渲染当前场景的听觉气氛。音乐文件较长，一般一次也只播放一首。音效作为游戏的增强表现，通常是一些较短的声音，例如子弹发射声音、刀剑打击声、炸弹爆炸等。音效支持一次同时播放多个。

本章内容：

- Cocos2d-JS中的音频
- 背景音乐何时播放
- 实例——给《保卫萝卜2》添加音频

6.1 Cocos2d-JS 中的音频

在JSB上，音频由`SimpleAudioEngine`类来处理。`SimpleAudioEngine`可以实现音频跨平台，其底层为Cocos2d-x自带的CocosDension库。而在JS层，音频统一由`cc.audioEngine`对象来控制。

6.1.1 `cc.audioEngine` 的常用 API

`cc.audioEngine`提供了简单易用的音频API，其控制音乐的常用API如表6-1和表6-2所示。

表6-1 `cc.audioEngine`控制音乐的常用API

返 回 值	函 数 名	说 明
无	`playMusic(url, loop)`	播放指定音乐，`loop`表示是否循环，默认值为`false`
无	`stopMusic(releaseData)`	停止音乐，`releaseData`为是否释放音乐，默认值为`false`
无	`pauseMusic()`	暂停播放音乐

（续）

返 回 值	函 数 名	说 明
无	`resumeMusic()`	恢复播放音乐
无	`rewindMusic()`	重新播放音乐
浮点数	`getMusicVolume()`	获取音乐音量，返回值范围`[0-1]`
无	`setMusicVolume(volume)`	设置音乐音量，音量设定值范围为`[0-1]`
布尔值	`isMusicPlaying()`	判断背景音乐是否正在播放

表6-2 **`cc.audioEngine`**控制音效的常用API

返 回 值	函 数 名	说 明
`object`	`playEffect(url, loop)`	播放指定音效，`loop`表示是否循环，返回值为`cc.Audio`对象
无	`setEffectsVolume(volume)`	设置音效音量，设定值范围为`[0-1]`
浮点型	`getEffectsVolume()`	获取音效音量，返回值范围`[0-1]`
无	`pauseEffect(audio)`	暂停指定音效
无	`pauseAllEffects()`	暂停所有音效
无	`resumeEffect(audio)`	恢复播放指定音效
无	`resumeAllEffects()`	恢复播放所有音效
无	`stopEffect(audio)`	停止指定音效
无	`stopAllEffects()`	停止所有音效
无	`unloadEffect(url)`	卸载指定音效

6.1.2 各平台下支持的音频格式

Cocos2d-JS是一个全平台游戏引擎，各平台支持的音频格式也有所不同，例如一个.ogg格式的音频在Android下能正常播放，而在iOS中则没有声音，这是因为iOS系统不支持.ogg音频格式解码的问题。表6-3和表6-4分别列出了各平台下对音乐和音效文件格式的支持情况。

表6-3 Cocos2d-JS在各平台支持的音乐格式

平台名称	支持类型	说 明
iOS	aac、caf、m4a、mp3和wav	可以播放AVAudioPlayer所支持的所有格式
Android	mid、ogg、mp3和wav	可以播放android.media.MediaPlayer所支持的所有格式
Web	oog和mp3	Web平台支持的格式依赖于浏览器
Windows	mid、mp3和wav	无

表6-4 Cocos2d-JS在各平台支持的音效格式

平台名称	支持类型	说 明
iOS	mp3、caf和m4a	可以播放CocosDesion引擎所支持的所有格式
Android	mp3、ogg和wav	对wav的支持不完美，推荐使用ogg
Web	mp3和wav	Web平台支持的格式依赖于浏览器
Windows	mid和wav	无

说明　音频格式的支持基本都依赖于系统层，CocosDesion底层的实现都是调用平台相关的API，具体可参见6.5节。

6.2　背景音乐何时播放

在3.4.3节中我们知道了，当SceneA切换到SceneB的时候，其场景生命周期相关函数的调用顺序如下：

```
1  SceneA : ctor
2  SceneA : onEnter
3  SceneA : onEnterTransitionDidFinish
4  SceneB : ctor
5  SceneA : onExitTransitionDidStart
6  SceneA : onExit
7  SceneB : onEnter
8  SceneB : onEnterTransitionDidFinish
```

当SceneA切换到SceneB的时候，Cocos2d-JS会先调用SceneB的`ctor`构造函数，然后继续调用SceneA的`onExitTransitionDidStart`和`onExit`函数，最后在SceneB的`onEnter`函数中将SceneB渲染出来。不难发现，SceneB的`ctor`构造函数是两个场景切换的关键所在。一般情况下，我们就是在这个函数中开始播放SceneB的背景音乐。只有在这个函数中播放背景音乐，才可以起到“人未到声先到”的过渡作用。

针对此问题，可以写一个测试工程。首先，需要声明两个场景，它们分别是SceneA和SceneB，并且重写其生命周期相关的函数，代码如下：

```
1  var SceneA = cc.Scene.extend({
2      ctor : function(){
3          this._super();
4          cc.log("SceneA : ctor");
5      },
6      onEnter : function () {
7          this._super();
8          cc.log("SceneA : onEnter");
9          // SceneA标签显示
10         var label = new cc.LabelTTF("This is SceneA.", "", 36);
11         this.addChild(label);
12         label.setPosition(cc.winSize.width / 2, cc.winSize.height / 2);
13         // 下一帧切换场景
14         var callback = cc.callFunc(function(sender){
15             var scene = new SceneB();
16             cc.director.runScene(scene);
17         }.bind(this));
18         var seq = cc.sequence(callback);
19         this.runAction(seq);
20     },
21     onEnterTransitionDidFinish : function () {
22         this._super();
```

```
23          cc.log("SceneA : onEnterTransitionDidFinish");
24      },
25      onExitTransitionDidStart : function () {
26          cc.log("SceneA : onExitTransitionDidStart");
27          this._super();
28      },
29      onExit : function () {
30          cc.log("SceneA : onExit");
31          this._super();
32      }
33 });
```

在SceneA的onEnter函数中定义了一个标签，用来显示当前的场景是SceneA，第13行到第20行代码的作用是让SceneA在下一帧的时候，自动切换到SceneB。SceneB的代码如下：

```
1  var SceneB = cc.Scene.extend({
2      ctor : function(){
3          this._super();
4          cc.log("SceneB : ctor");
5          // 播放背景音乐
6          cc.audioEngine.playMusic("res/unit07_audio/music/main_music.mp3");
7      },
8      onEnter : function () {
9          this._super();
10         cc.log("SceneB : onEnter");
11         // SceneB标签显示
12         var label = new cc.LabelTTF("This is SceneB.", "", 36);
13         this.addChild(label);
14         label.setPosition(cc.winSize.width / 2, cc.winSize.height / 2 - 100);
15         // 创建15000个精灵，故意造成卡顿
16         for (var i = 0; i < 15000; i++){
17             var node = new cc.Sprite(res.sh_node_128_png);
18             this.addChild(node);
19             node.setPosition(1.2 * i, V.h2);
20         }
21     },
22     onEnterTransitionDidFinish : function () {
23         this._super();
24         cc.log("SceneB : onEnterTransitionDidFinish");
25     },
26     onExitTransitionDidStart : function () {
27         cc.log("SceneB : onExitTransitionDidStart");
28         this._super();
29     },
30     onExit : function () {
31         cc.log("SceneB : onExit");
32         this._super();
33     }
34 });
```

在SceneB的ctor函数中播放背景音乐，然后在onEnter函数中（上述代码15~20行）创建了15000个精灵，使场景卡顿一段时间，这主要用来模拟加载SceneB资源。运行SceneA，立马可以听到SceneB的背景音乐，而这个时候，还未进入SceneB中的onEnter函数，所以SceneB中的15000

个精灵还未被渲染出来，这就达到了“人未到声先到”的过渡效果。

6.3　实例——给《保卫萝卜 2》添加音频

在本章中，我们将给《保卫萝卜2》添加部分音乐和音频。

前面说到，背景音乐正确的播放时机应该是在场景的构造函数中，所以我们应该在MainMenuScene的ctor构造函数中播放对应的背景音乐，代码如下：

```
1 var MainMenuScene = cc.Scene.extend({
2     backgroundLayer : null,
3     mainLayer : null,
4     ctor : function(){
5         this._super();
6         cc.audioEngine.playMusic(res.sd_mm_BGMusic_mp3, true);
7     }
8     // ......
9 });
```

然后，也给主页面中的“天天向上”和“开始冒险”菜单按钮添加了对应的点击音效，它们在MMMainLayer类中，代码如下：

```
1 // 加载“开始冒险”和“天天向上”菜单
2 loadMenu : function(){
3     // ......
4     // 开始冒险
5     var start = new cc.MenuItemSprite(
6         startNormal,
7         startPress,
8         startDisabled,
9         function(){
10            cc.log(“点击‘开始冒险’按钮”);
11            cc.audioEngine.playEffect(res.sd_mm_Select_mp3);
12        }.bind(this));
13    start.setPosition(V.w2 - 8, V.h2 + 75);
14    // ......
15 }
```

6.4　小结

通过本章的学习，我们了解了如何在游戏中控制音乐和音频文件的播放，熟悉了各个平台下所支持的音乐以及音效格式，并且清楚了应该在下一个场景的构造函数中播放背景音乐，这样可以起到“人未到声先到”的过渡效果，增强了游戏体验。最后，我们给《保卫萝卜2》的主页面添加了背景音乐。

6.5 参考资源

本章的参考资源如下。

- 官方API：http://api.cocos.com/cn/d0/d75/classcocos2d_1_1experimental_1_1_audio_engine.html。
- iOS音频使用：https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/MultimediaPG/UsingAudio/UsingAudio.html。
- Android媒体支持格式：http://developer.android.com/guide/appendix/media-formats.html。

第 7 章 屏幕适配

一直以来，屏幕适配让开发者甚为烦躁，特别是在Android设备泛滥或“碎片化”的今天，市面上的Android设备有各种屏幕尺寸以及分辨率。据OpenSignalMaps发布的Android碎片化统计报告数据显示，在2012年，支持Android的设备共有3997种，2013年发展到11868种，到了2014年高达18796种，2015年还在持续增长中。再加上iPhone 6s、iPhone 6s Plus以及iPad Pro等的推出，移动设备的屏幕尺寸更是越来越多，要想将游戏都适配到这么多屏幕上，并不是一件简单的事情。

本章内容：

- 屏幕适配原理
- 如何使用适配模式
- 系统预设适配模式
- 推荐的适配方案

7.1 屏幕适配原理

Cocos引擎在Cocos2d-x 2.0.4版本时推出了多屏幕适配方案，提出了设计分辨率（design resolution）的概念，它屏蔽了资源分辨率（resource resolution）和屏幕分辨率（screen resolution）。开发者只需将节点在设计分辨率上进行位置设定即可。

设计分辨率指的是游戏在开发时所指定的设计分辨率尺寸，在main.js的`cc.game.onStart()`函数中设定。下列代码设定设计分辨率的宽为1136，高为640：

```
1  cc.view.setDesignResolutionSize(1136, 640, cc.ResolutionPolicy.SHOW_ALL);
```

资源分辨率指的是图片资源的尺寸，图7-1所示的图片的资源分辨率为1334 × 750。

屏幕分辨率为屏幕图像的精密度，指显示屏所能显示的像素有多少，显示屏可显示的像素越多，画面就越精细。不同的移动设备有不同的屏幕分辨率，在显示屏大小固定时，屏幕分辨率越高，图像越清晰。例如，主流移动设备iPhone 6s的屏幕分辨率为1334 × 750，小米4的屏幕分辨率为1920 × 1080。

除了这3个分辨率的概念之外，屏幕适配的背后又存在两个转换适配过程，分别是从资源分

辨率转换到设计分辨率，以及从设计分辨率转换成屏幕分辨率。



图7-1　图片资源分辨率

Cocos2d-JS游戏在进行屏幕适配的时候，有可能会先将资源分辨率转换成设计分辨率，这取决于你是否设定了内容缩放因子。内容缩放因子指的是资源分辨率的宽比设计分辨率的宽，又或者是资源分辨率的高比设计分辨率的高，即RW / DW 或者 RH / DH，其中RW、DW等为简写。为了描述方便，在本章后面，我都将采用简写的方式表达，简写对应的全称如表7-1所示。

表7-1　简写对应的全称

全　　称	简　　称	说　　明
Resources Width	RW	资源分辨率的宽度
Resources Height	RH	资源分辨率的高度
Design Width	DW	设计分辨率的宽度
Design Height	DH	设计分辨率的高度
Scene Width	SW	屏幕分辨率的宽度
Scene Height	SH	屏幕分辨率的高度

内容缩放因子一般也是在`cc.game.onStart()`函数中设定的，代码如下：

```
1  cc.director.setContentScaleFactor(768 / 640);
```

用高度比作为内容缩放因子，保证了资源图片的垂直方向在设计分辨率范围内的全部显示。用宽度比作为内容缩放因子，保证了资源图片的水平方向在设计分辨率范围内的全部显示。

接下来要进行第二次转换，就是将设计分辨率转换成屏幕分辨率。Cocos2d-JS一共为第二次转换提供了5种适配方式，这里称为适配模式。只有进行第二次转换后，才是最终显示在屏幕上的效果。关于这5种适配模式，将在7.3节中详细介绍。

7.2　如何使用适配模式

了解第二次转换的5种适配模式前，先要了解一下Cocos2d-JS中是如何使用适配模式的。在main.js文件中的cc.game.onStart方法里可以看到如下接口：

```
1  cc.view.setDesignResolutionSize(width,height,resolutionPolicy);
```

此代码用来设置设计分辨率的大小和适配模式，其中width和height分别表示游戏设计分辨率的宽度和高度，resolutionPolicy为5种适配模式之一，其可选值为如下5种（我们将在后面进一步介绍这5个选项）：

```
1  cc.ResolutionPolicy.EXACT_FIT
2  cc.ResolutionPolicy.NO_BORDER
3  cc.ResolutionPolicy.SHOW_ALL
4  cc.ResolutionPolicy.FIXED_HEIGHT
5  cc.ResolutionPolicy.FIXED_WIDTH
```

另外，你也可以通过如下接口修改适配模式：

```
1  cc.view.setResolutionPolicy(resolutionPolicy);
```

在Web上，浏览器的大小可能随时被改变，例如通过用户拖曳来改变浏览器大小，或者当玩家转动手机方向的时候。适配方案允许在浏览器大小变化的时候重新进行屏幕适配，其接口如下：

```
1  cc.view.resizeWithBrowserSize(isEnabled);
```

其中参数isEnabled为Boolean类型的值，表示是否开启重新适配。

说明　cc.view.resizeWithBrowserSize(isEnabled)函数仅在Web上有效。

为了更灵活地应对变化，Cocos2d-JS为cc.view的大小变化添加了一个回调函数，其接口如下：

```
1  cc.view.setResizeCallback(callback);
```

其中callback为一个回调函数。例如，在微信浏览器上，玩家转动手机方向的时候，你便可以通过此函数判断当前手机是横屏还是竖屏，示例代码如下：

```
1  var callback = function(){
2      if (cc.winSize.width < cc.winSize.height) {
3          cc.log("屏幕改变为：横屏");
4      }else if (cc.winSize.width > cc.winSize.height){
5          cc.log("屏幕改变为：竖屏");
6      }else{
7          cc.log("屏幕方向没有变化");
8      }
9
10     cc.log("旋转屏幕前 : ",cc.winSize);
11     var fun = cc.callFunc(function(){
12         cc.log("旋转屏幕后：", cc.winSize);
13     });
14     cc.director.getRunningScene().runAction(fun);
```

```
15 };
16 cc.view.setResizeCallback(callback);
```

需要特别注意的是，第2行代码中的判断条件是`cc.winSize.width < cc.winSize.height`，即如果画布的宽度小于高度，则是横屏。这看上去似乎是错误的，但是实际上这段代码是正确的，因为`callback`回调函数在重新适配屏幕之前就被调用了，所以此时的`cc.winSize`的值还是上一次屏幕适配时的值。第10行代码打印出当前的`cc.winSize`的值，然后在下一帧的时候，再次打印出`cc.winSize`的值，Chrome浏览器控制台输出结果如图7-2所示。

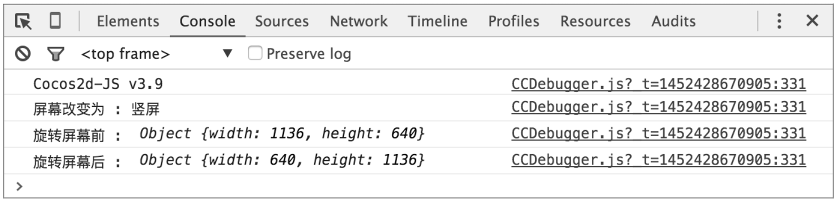


图7-2　屏幕旋转前后`cc.winSize`的值

另外，Chrome浏览器自身携带了一个手机模拟器，在Mac OS X操作系统下，可以通过快捷键shift+command+M（Windows对应Ctrl+Shift+M）打开，如图7-3所示，你可以在此模拟器中选择市面上常见的手机分辨率，也可以自定义分辨率，这可以让你快速查看到你的游戏在不同分辨率下的适配效果。



图7-3　Chrome浏览器中的手机模拟器

在Chrome浏览器中的手机模拟器中调试你的游戏时，若首次点击屏幕，游戏会自动全屏，你可以在`cc.game.onStart`函数的开头关闭自动全屏，代码如下所示：

```
1  cc.game.onStart = function(){
2      // ......
3      cc.view.enableAutoFullScreen(false); //关闭游戏在浏览器上自动全屏
4      cc.view.enableRetina(cc.sys.os === cc.sys.OS_IOS ? true : false);
5      // ......
6  };
```

7.3　系统预设适配模式

之前我曾提到Cocos2d-JS中预设了5种适配模式来进行第二次转换，下面将详细介绍每种适配模式的效果。

7.3.1　cc.ResolutionPolicy.SHOW_ALL

cc.ResolutionPolicy.SHOW_ALL表示显示所有，此模式会按原始宽高比等比缩放游戏世界以适配屏幕，保证了最小的一个方向上充满屏幕，另一个方向等比缩放，从而使得游戏内容全部可见。当然，这样当屏幕宽高比不等于设计分辨率的宽高比时，窗口中会有一定的黑边，如图7-4所示。



图7-4　cc.ResolutionPolicy.SHOW_ALL适配模式的效果

说明　图7-3中背景图的资源分辨率大小为1024 × 768，本章后面的图也都将以此图为基础，并且设计分辨率也为1024 × 768，屏幕分辨率为1136 × 640。

7.3.2　cc.ResolutionPolicy.NO_BORDER

cc.ResolutionPolicy.NO_BORDER表示没有边框，此模式也会按原始宽高比等比缩放游戏世界以适配屏幕，但实际上，cc.ResolutionPolicy.NO_BORDER与cc.ResolutionPolicy.

SHOW_ALL模式是相反的关系，cc.ResolutionPolicy.NO_BORDER并不会留有黑边，所以屏幕分辨率宽高比不同于设计分辨率的宽高比时，游戏世界会被部分切割在屏幕外，如图7-5所示。



图7-5　cc.ResolutionPolicy.NO_BORDER模式的适配效果

另外，在这种情况下，cc.visibleRect代表的就是Canvas在游戏世界中的视窗，其大小比cc.winSize要小一些。我将cc.visibleRect和cc.winSize打印出来，得到的结果如图7-6所示，可以发现，此时cc.visibleRect的height不是768，而是575.712143928036。

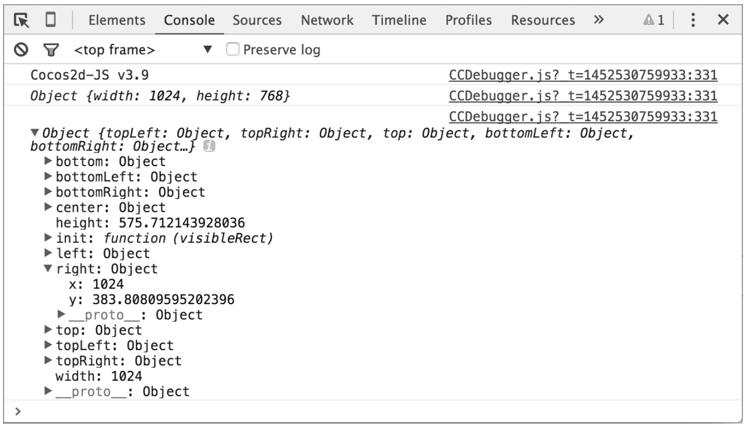


图7-6　cc.winSize和cc.visibleRect的值

7.3.3　cc.ResolutionPolicy.EXACT_FIT

cc.ResolutionPolicy.EXACT_FIT表示拉伸，此模式保证了游戏世界完全铺满屏幕，但是可能会出现图像拉伸。图7-7 为cc.ResolutionPolicy.EXACT_FIT模式的适配效果，可以明显看出，横向被拉伸了。



图7-7　cc.ResolutionPolicy.EXACT_FIT模式的适配效果

7.3.4　cc.ResolutionPolicy.FIXED_HEIGHT

cc.ResolutionPolicy.FIXED_HEIGHT表示固定高度，此模式会纵向放大游戏世界以适应屏幕的宽度，横向按原始宽高比放大，它保证了游戏内容不会被拉伸，如图7-8所示。



图7-8　cc.ResolutionPolicy.FIXED_HEIGHT适配模式的效果

cc.ResolutionPolicy.FIXED_HEIGHT模式下，游戏世界坐标系等同于整个屏幕（Web中对应Canvas元素）的坐标系，即始终以屏幕的左下角为坐标原点，这和cc.ResolutionPolicy.SHOW_ALL模式不同。在图7-4中可以看到，cc.ResolutionPolicy.SHOW_ALL模式下FPS显示的位置并不是正左下角，而是向右有所偏离，但是cc.ResolutionPolicy.FIXED_HEIGHT模式下的游戏坐标始终为正左下角。所以，可以得出，在cc.ResolutionPolicy.FIXED_HEIGHT模式下的游戏可显示内容区域实际上比cc.ResolutionPolicy.SHOW_ALL模式更多。

7.3.5 cc.ResolutionPolicy.FIXED_WIDTH

cc.ResolutionPolicy.FIXED_WIDTH表示固定宽度，此模式会横向放大游戏世界以适屏幕的宽度，纵向按原始宽高比放大，如图7-9所示。cc.ResolutionPolicy.FIXED_WIDTH和cc.ResolutionPolicy.FIXED_HEIGHT仅仅只是相反关系，前者固定高度，后者固定宽度。



图7-9 cc.ResolutionPolicy.FIXED_WIDTH适配模式的效果

7.4 推荐的适配方案

cc.ResolutionPolicy.NO_BORDER曾经是官方推荐使用的适配方案，它不会拉伸图像，同时在一个方向上撑满了屏幕，但是后来新加入的cc.ResolutionPolicy.FIXED_HEIGHT和cc.ResolutionPolicy.FIXED_WIDTH策略撼动了它的地位。目前，这两种策略也是官方推荐使用的方案。

cc.ResolutionPolicy.FIXED_HEIGHT 和 cc.ResolutionPolicy.FIXED_WIDTH 策略适配的用法非常简单，甚至可以提取出公式，如下：

```
1  // 横屏游戏
2  cc.view.setDesignResolutionSize(DW, DH, cc.ResolutionPolicy.FIXED_HEIGHT);
3  cc.director.setContentScaleFactor(RH / DH);
4  // 竖屏游戏
5  cc.view.setDesignResolutionSize(DW, DH, cc.ResolutionPolicy.FIXED_WIDTH);
6  cc.director.setContentScaleFactor(RW / DW);
```

在实际开发中，我们一般用到的也就是这两种模式，横屏游戏使用cc.ResolutionPolicy.FIXED_HEIGHT模式固定高度，竖屏游戏使用cc.ResolutionPolicy.FIXED_WIDTH模式固定宽度。

以横屏游戏《保卫萝卜2》为例（竖屏同理），它采用的模式是cc.ResolutionPolicy.FIXED_HEIGHT，并且它直接将游戏设计分辨率设置为和资源（游戏背景图）分辨率一样大，即资源分辨率和设计分辨率都为1136 × 640，而内容缩放因子保持默认值1。因为在cc.ResolutionPolicy.FIXED_HEIGHT模式下，1136×640大小的内容能铺满960×640的屏幕，

而960 × 640大小的背景图并不能铺满1136 × 640大小的屏幕，所以在横屏模式下，你应当尽可能地保证你的游戏背景图片的宽度足够大，例如在960 × 640 和1136 × 640这两种尺寸中，你应该考虑后者而不是前者。

1136 × 640 的尺寸为16∶9，从iPhone 5开始到iPhone 6s Plus，屏幕比例也都是16∶9，那么选择iPhone 5s的1136 × 640还是iPhone 6s的1334 × 750实际上都是一样的，这取决于你自己。

另外要说明的是，在开发中，节点的坐标要懂得灵活选择参考系，例如《保卫萝卜2》主页面中的设置按钮是以屏幕中心点为坐标参照点，而不是屏幕的左下角。这样的做法保证了其在各种屏幕分辨率下节点的布局都不会发生变化，只是显示的区域大小不同而已。这也就是说，在`cc.ResolutionPolicy.FIXED_HEIGHT`或者`cc.ResolutionPolicy.FIXED_WIDTH`模式下，你应当尽可能地将游戏元素集中在屏幕中间，左右（或上下）两侧在比较窄（对比1136 × 640）的屏幕分辨率上（例如960 × 640）可能不会显示出来。《保卫萝卜2》在iPhone 6和iPhone 4上的运行效果如图7-10和图7-11所示。



图7-10　《保卫萝卜2》主页面适配在iPhone 6上的效果



图7-11　《保卫萝卜2》主页面适配在iPhone 4上的效果

与图7-10相比，图7-11只是左右被“裁剪”了一部分，其他的（例如节点位置等）并没有发生变化。这是因为，主页面上的节点，都是以屏幕正中心为坐标参考点，但是，在其他的情况下就可能发生变化。例如，图7-12为《保卫萝卜2》的关卡选择页面，左上角有个返回主页面的按钮，就应该以屏幕左上角为参考点，坐标设定代码如下：

```
1  button.setPosition(posX, cc.winSize.height - offsetY);
```

其中，offsetY为Y轴向下偏移量。而右上角的生命星则应当以右上角为坐标参考点，它的坐标设定代码如下：

```
1  star.setPosition(cc.winSize.width - offsetX,  cc.winSize.height - offsetY);
```



图7-12 《保卫萝卜2》关卡选择页面

另外，虽然市面上移动设备的屏幕五花八门，但是在做适配测试的时候，可以考虑借鉴“极值”思想。以iOS设备为例，若游戏能适配在屏幕“最长”的iPhone 6s（分辨率1334 × 750）和“最短”的iPad（分辨率1024 × 768），那么屏幕分辨率夹在这两者之间的，便都可以拥有很好的适配效果。

7.5 小结

本章主要介绍了Cocos引擎适配的过程，以及预设的5种适配模式：cc.ResolutionPolicy.SHOW_ALL、cc.ResolutionPolicy.NO_BORDER、cc.ResolutionPolicy.EXACT_FIT、cc.ResolutionPolicy.FIXED_WIDTH和cc.ResolutionPolicy.FIXED_HEIGHT。Cocos2d-JS默认使用的是cc.ResolutionPolicy.SHOW_ALL模式，而目前推荐的适配方式是如果游戏是横屏的，那么使用cc.ResolutionPolicy.FIXED_HEIGHT模式；如果是竖屏的，则使用cc.ResolutionPolicy.FIXED_WIDTH模式。

7.6　参考资源

本章的参考资源如下。

- Cocos2d-x多分辨率适配完全解析 : http://www.tairan.com/archives/4809/。
- Cocos2d-JS的屏幕适配方案：http://www.cocos.com/doc/article/index?type=cocos2d-x&url=/doc/cocos-docs-master/manual/framework/cocos2d-js/4-essential-concepts/4-4-resolution-policies/zh.md。

第二部分

进　阶　篇

第 8 章 数据存储

对于计算机来说，所有的东西都只是一堆数据，这些数据可以是简单的数字，也可以是文字、图像、声音、视频等。但相对游戏开发而言，我们平常所说的数据，更多的是偏向玩家的数据，例如玩家拥有的金币、钻石，又或者是用户的一些配置信息，例如是否保存账号、声音开关等。因为这些数据不受游戏运行状态的影响，所以它们通常保存在本地文件中，这也就是所谓的数据存储，又称数据持久化。

在Cocos2d-JS中，数据存储可以通过`cc.sys.localStorage`、plist、JSON以及SQLite等途径实现，本章将围绕这些实现来展开。

本章内容：

- `cc.sys.localStorage`
- JSON文件读取
- plist文件读取
- SQLite实现
- 实例——解锁《保卫萝卜2》的“天天向上”玩法

8.1 cc.sys.localStorage

在实际开发中，我们通常会把用户设置、是否第一次开启游戏等这些简单而又单独的数据通过`cc.sys.localStorage`存储。`cc.sys.localStorage`采用键值对的设计方式，其API如表8-1所示。

表8-1 `cc.sys.localStorage` API

函　　数	说　　明
`cc.sys.localStorage.setItem(key, value);`	key为字符串类型，value为数值或字符串
`cc.sys.localStorage.getItem(key);`	返回值为数值或者字符串。若数据不存在，则返回null
`cc.sys.localStorage.removeItem(key);`	根据键删除值
`cc.sys.localStorage.clear();`	清空所有通过cc.sys.localStorage保存的数据

在HTML5上，因为cc.sys.localStorage的实现依赖于windows.localStorage，所以通过cc.sys.localStorage存储的数据并不安全，它可能很容易被一些例如电脑管家等第三方软件清除掉，从而存在安全隐患。而在JSB中，cc.sys.localStorage底层则由SQLite3实现，Cocos2d-x JSB封装了一些简单的调用函数，然后做了脚本绑定（见第16章），具体代码在cocos2d-x/cocos/storage/local-storage路径下。

综上所述，除非是一些简单并且不在乎其安全问题的数据，否则我们一般建议放在服务端。另外，在开发过程中，除了通过调用cc.sys.localStorage.clear()来清除LocalStorage中的数据外，还可以在Chrome浏览器中清空“Cookie以及其他网站数据和插件数据”来达到目的，如图8-1所示。



图8-1　Cookie以及其他网站数据和插件数据

技巧　在HTML5上，可以通过cc.log(cc.sys.localStorage)的方式，打印出localStorage里所有的数据。

8.2　JSON 文件读取

JSON（JavaScript Object Notation）是一种轻量级的数据交互格式，它是JavaScript原生格式，以键值对的形式存在，如图8-2所示。通常情况下，JSON主要用于客户端和服务端之间的数据传送，或者是当作本地配置文件，就像Cocos2d-JS工程下的project.json文件那样。

JSON的读取较为简单，在Cocos2d-JS中可以通过如下方式读取：

```
1  var jsonArray = [ // 定义要加载的JSON文件数组
2      "res/unit08_data/config.json"
3  ];
4  // 加载JSON文件，可以批量加载，读取出来的数据保存在results数组中
5  cc.loader.load(jsonArray, function(err, results){
6      if(err){
7          cc.error("Failed to load %s, %s .", jsonArray); // 错误提示
8          return;
```

```
9       }
10      cc.log(results);     // 打印[所有加载结果]
11      cc.log(results[0]); // 打印[读取的第一个JSON]
12 });
```



```
{
    "project_type": "javascript",

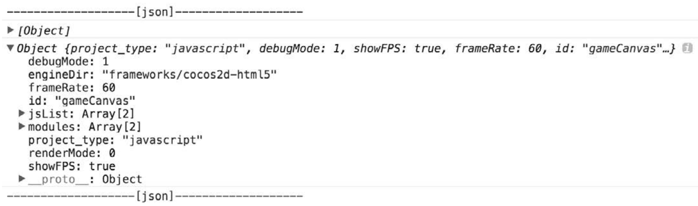
    "debugMode" : 1,
    "showFPS" : true,
    "frameRate" : 60,
    "id" : "gameCanvas",
    "renderMode" : 0,
    "engineDir":"frameworks/cocos2d-html5",

    "modules" : ["cocos2d", "extensions"],

    "jsList" : [
        "src/resource.js",
        "src/config.js"
    ]
}
```

图8-2　JSON格式示意图

控制台打印出来的JSON数据如图8-3所示。



```
-------------------[json]-------------------
▶ [Object]
▼ Object {project_type: "javascript", debugMode: 1, showFPS: true, frameRate: 60, id: "gameCanvas"…}
    debugMode: 1
    engineDir: "frameworks/cocos2d-html5"
    frameRate: 60
    id: "gameCanvas"
  ▶ jsList: Array[2]
  ▶ modules: Array[2]
    project_type: "javascript"
    renderMode: 0
    showFPS: true
  ▶ __proto__: Object
-------------------[json]-------------------
```

图8-3　控制台打印出JSON的格式信息

8.3　plist 文件读取

plist文件通常用于存储用户设置，也可以用于存储一些成组的配置信息，例如粒子系统等都是采用plist文件存储数据。plist格式众多，其主流方式为XML，如图8-4所示。



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>debugMode</key>
    <integer>1</integer>
    <key>engineDir</key>
    <string>frameworks/cocos2d-html5</string>
    <key>frameRate</key>
    <integer>60</integer>
    <key>id</key>
    <string>gameCanvas</string>
    <key>jsList</key>
    <array>
        <string>"src/config.js",</string>
        <string>"src/resource.js"</string>
    </array>
    <key>modules</key>
    <array>
        <string>cocos2d</string>
        <string>extensions</string>
    </array>
    <key>project_type</key>
    <string>javascript</string>
    <key>renderMode</key>
    <integer>0</integer>
    <key>showFPS</key>
    <true/>
</dict>
</plist>
```

图8-4　plist格式示意图

plist文件的读取方式和JSON一致，其代码如下：

```
1  var plistArray = [ // 定义要加载的plist文件数组
2      "res/unit08_data/config.plist"
3  ];
4  // 加载plist文件，可以批量加载，读取出来的数据保存在results数组中
5  cc.loader.load(plistArray, function(err, results){
6      if(err){
7          cc.error("Failed to load %s, %s .", plistArray); // 错误提示
8          return;
9      }
10     cc.log(results);      // 打印[所有加载结果]
11     cc.log(results[0]); // 打印[读取的第一个plist]
12 });
```

控制台打印出来的plist数据如图8-5所示。

```
-------------------[plist]-------------------
▶[Object]
▼Object {project_type: "javascript", debugMode: 1, showFPS: true, frameRate: 60, id: "gameCanvas"…}
   debugMode: 1
   engineDir: "frameworks/cocos2d-html5"
   frameRate: 60
   id: "gameCanvas"
  ▶jsList: Array[2]
  ▶modules: Array[2]
   project_type: "javascript"
   renderMode: 0
   showFPS: true
  ▶__proto__: Object
-------------------[plist]-------------------
```

图8-5 控制台打印出plist的格式信息

从图8-3和图8-4可以看出，JSON文件格式较为简单，而plist文件的格式较为烦琐。一般情况下，JSON文件可以像平常写代码那样，直接手动编写。而plist文件则由plist编辑器去做。在Mac OS下，可以通过Xcode、PlistEdit Pro等工具。在Windows系统中，也有plist Editor等一系列优秀工具。关于更多的plist编辑器，读者可自行网上搜索。

8.4 SQLite 实现

相比plist和JSON而言，SQLite可以给我们带来更安全、更强大的数据存储，但其使用以及学习成本也更高了。

虽然在JSB上，`cc.sys.localStorage`底层由SQLite3实现，但是Cocos2d-JS本身没有将SQLite绑定到JavaScript上，也就是说，我们无法在JavaScript层上指定要执行的SQLite3数据库语句。不过，我们可以通过代码绑定的方式自行实现，相关内容可参见第16章的章节实例。

8.5 实例——解锁《保卫萝卜2》的“天天向上”玩法

实际上，在《保卫萝卜2》主页面中，“天天向上”按钮是锁住的，如图8-6所示。

若“天天向上”按钮是“锁住”状态，则点击“天天向上”按钮，会出现如图8-7所示的对话框。图8-7所示的功能本质上是一个内购，因为内购需要接入第三方SDK，且可选的SDK数量繁多，每个平台的接入也都不一样，所以这里我将这个功能简化，采用本地数据存储的方式作为本章的实例。



图8-6 “天天向上”按钮锁



图8-7 “天天向上”解锁对话框

基于上一章的实例，我给当前的Layer定义了一个isUpUnlock属性，表示是否解锁了“天天向上”，其默认值为false。此外，又添加了一个loadConfig()函数，用来从localStorage中获取isUpUnlock的值，代码如下：

```
1  // 加载[配置]
2  loadConfig : function(){
3      // 是否已经解锁了“天天向上”
4      this.isUpUnlock = cc.sys.localStorage.getItem(Config.IS_UP_UNLOCK_KEY) || "NO";
5  }
```

第4行代码通过cc.sys.localStorage获取出“是否已经解锁”的值，Config.IS_UP_UNLOCK_KEY是一个字符串变量。值得特别注意的是，因为cc.sys.localStorage在JSB的底层实现是SQLite3，并不能像浏览器对象中的LocalStorage那样可以直接存放布尔类型的值，所以这里用字符串类型代替。若cc.sys.localStorage取出的值为null，则让this.isUpUnlock的值为NO。另外，我在Carrot/src路径下新建了一个define.js文件，而Config对象就是定义在此文件中，代码如下：

```
1  // 配置对象
2  var Config = {};
3  Config.IS_UP_UNLOCK_KEY = "is_up_unlock_key";
```

然后在“加载菜单”按钮的方法中，给“天天向上”按钮加上锁的精灵，代码如下：

```
1  // 加载[“开始冒险”和“天天向上”菜单]
2  loadMenu : function(){
3      // ......
4      if (this.isUpUnlock === "NO") {
5          var upLock = new cc.Sprite("res/MainMenu/front_btn_floor_locked.png");
6          floor.addChild(upLock);
7          this.upLock = upLock;
8          upLock.setPosition(floor.width + 5, floor.height / 2 + 25);
```

```
9       }
10      // ......
11 }
```

我为当前的Layer定义了一个upLock属性来指向这个锁，这是因为在后面解锁“天天向上”时，可以通过调用this.upLock.removeFromParent()函数来将锁从当前Layer移除掉。接着在“天天向上”按钮的点击事件中，键入如下代码：

```
1  var floor = new cc.MenuItemSprite(
2      floorNormal,
3      floorPress,
4      floorDisabled,
5      function(){
6          cc.audioEngine.playEffect(res.sd_mm_Select_mp3);
7          if (this.isUpUnlock == "NO") {
8              // 分发事件[打开解锁面板]
9              cc.eventManager.dispatchEvent(new cc.EventCustom(jf.EventName.OPEN_
               UNLOCK_UP_LAYER));
10         } else {
11             cc.log("TODO: 实现“天天向上”按钮的功能");
12         }
13     }.bind(this));
```

在上述的代码中，第7行代码用于判断“天天向上”是否已经解锁了，若未解锁，则分发一个“打开解锁面板”的事件。很显然，事件的设计是发布者/订阅者的设计模式。那么，我们在MainMenuScene中实现“打开解锁面板”的事件监听。当“打开解锁面板”事件被触发时，MainMenuScene便会接收到这一事件，从而做“打开解锁面板”的处理，代码如下：

```
1  registerEvent : function(){
2      var listener = cc.EventListener.create({
3          event       : cc.EventListener.CUSTOM,
4          target      : this,
5          eventName   : jf.EventName.OPEN_UNLOCK_UP_LAYER,
6          callback    : this.onLoadUnlockLayer
7      });
8      cc.eventManager.addListener(listener, this);
9  },
10 onLoadUnlockLayer : function(event){
11     var target = event.getCurrentTarget();  // 获取当前事件监听器的事件源
12     target.unlockLayer = new MMUnlockLayer();
13     target.addChild(target.unlockLayer);
14 }
```

在第5行代码中，事件名字jf.EventName.OPEN_UNLOCK_UP_LAYER也定义在define.js中，代码如下：

```
1  var jf = {};         // 定义自己的命名空间
2  jf.EventName = {};  // 事件名字对象
3  jf.EventName.OPEN_UNLOCK_UP_LAYER = "open_unlock_up_layer";
4  jf.EventName.UNLOCK_UP = "unlock_up";
```

当名字为jf.EventName.OPEN_UNLOCK_UP_LAYER的自定义事件被触发时，MainMenu

Scene会调用onLoadUnlockLayer()函数，从而创建出“解锁天天向上”对话框，并将其添加到当前场景中。

MMUnlockLayer为一个LayerColor，其颜色为黑色，不透明度为220，这样可以“高亮突出”当前层里面的内容。MMUnlockLayer里的内容是一个对话框，很显然，它属于UI控件。UI控件在第10章中会讲到，若此时你看不太懂下面要编写的代码，那也无妨，你可以在阅读第10章之后，再回过头来阅读本章。下面开始MMUnlockLayer。

首先，若工程要用到UI模块，则需要在project.json中的modules数组中添加"ccui"，如"modules" : ["cocos2d", "ccui"]。然后，在MMUnlockLayer中，我们需要先创建一个Layout来存放这些UI控件，代码如下：

```
1  // 加载[Layout]
2  loadLayout : function(){
3      var node = new ccui.Layout();
4      this.addChild(node);
5      this.layout = node;
6      node.setContentSize(cc.winSize);// 设置内容大小
7      node.setTouchEnabled(true);     // 开启触摸
8  }
```

Layout的默认大小是cc.size(0, 0)，所以在第6行中，我设置了当前Layout的大小为cc.winSize。然后在第7行代码中开启了触摸事件，这样做是为了“吞噬”掉UI的触摸事件，使得MMMainLayer中的“开始冒险”和“天天向上”组成的菜单不会被点击到。

接着，又定义了一个loadBackground()函数来创建如图8-8所示的背景图片，代码如下：

```
1  // 加载[背景]
2  loadBackground : function(){
3      var node = new ccui.ImageView("res/Common/bg/woodbg_notice.png");
4      this.layout.addChild(node);
5      this.background = node;
6      node.setPosition(cc.winSize.width / 2, cc.winSize.height / 2);
7  }
```

将背景添加到Layout中，因为这是一个UI模块。并且，我为当前的Layer定义了一个background属性，指向了当前的背景，这是因为后面要开发的UI控件都需要以background为父节点。



图8-8　对话框背景

图8-7中所示的锁、解锁所需的价钱、文本提示等已经做成了一张图片，它的创建方式和背景是一样的，这里不贴出，读者可自行阅读随书源码。

剩下的两个按钮的开发也都大同小异，下面以“确定”按钮为例进行介绍。创建“确定”按钮的代码如下：

```
1  // 加载[“确定”按钮]
2  loadConfirmButton : function(offsetX, posY){
3      var node = new ccui.Button();
4      this.background.addChild(node);
5      node.loadTextures("res/UI/btn_blue_m.png", "res/UI/btn_blue_m_pressed.png",
       "");
6      node.setPosition(this.background.width / 2 - node.width / 2 - offsetX, posY);
7      node.addTouchEventListener(function(sender, type){
8          switch (type) {
9              case ccui.Widget.TOUCH_ENDED:
10                 var event = new cc.EventCustom(jf.EventName.UNLOCK_UP);
11                 event.setUserData({
12                     isSuccess : true
13                 });
14                 cc.eventManager.dispatchEvent(event);
15                 this.removeFromParent();
16                 break;
17         }
18     }, this);
19     // “确定”按钮的文字图片
20     var infoNode = new ccui.ImageView("res/UI/zh/btn_blue_m_ok.png");
21     node.addChild(infoNode);
22     infoNode.setPosition(node.width / 2, node.height / 2);
23 }
```

说明 UI控件有一套自己的触摸事件机制，并不需要我们像第5章中所讲的那样给每个控件添加事件监听器。

第5行代码指定了按钮的三个状态的纹理，分别是正常状态、选中状态、禁用状态的纹理。因为在此对话框中并没有禁用此按钮的需求，所以我给`loadTexture`函数的最后一个参数传递了一个空字符串。接着，第7行代码给“确定”按钮添加了触摸事件，当触摸抬起的时候，分发了一个“天天向上解锁”的事件，事件携带着“解锁成功”的数据。最后，第15行代码执行了从父节点删除当前层的操作。

很显然，若“天天向上”解锁成功，则主页面中“天天向上”菜单项中的锁就应该移除掉，所以应该在`MMMainLayer`中实现“天天向上解锁”的事件监听，代码如下：

```
1  registerEvent : function(){
2      var listener = cc.EventListener.create({
3          event       : cc.EventListener.CUSTOM,
4          target      : this,
5          eventName   : jf.EventName.UNLOCK_UP,
```

```
6           callback    : this.onUnLockUp
7       });
8       cc.eventManager.addListener(listener, this);
9   },
10 // 解锁“天天向上”
11 onUnLockUp : function(event){
12      var target = event.getCurrentTarget();
13      var data = event.getUserData();
14      if (data.isSuccess !== undefined && data.isSuccess) {
15          // 数据保存[解锁成功]
16          cc.sys.localStorage.setItem(Config.IS_UP_UNLOCK_KEY, "YES");
17          target.isUpUnlock = true;
18          target.upLock.removeFromParent();
19      }
20 }
```

当“天天向上”解锁成功后，第16行代码将解锁成功的状态进行数据持久化。

至此，“天天向上”解锁功能开发完毕。运行项目，点击“天天向上”按钮，在弹出的对话框中点击“确定”按钮，便可解锁成功。当关闭游戏之后，第二次再启动游戏时，会发现“天天向上”菜单按钮上并不会出现锁了。

8.6 小结

通过本章的学习，我们学习了Cocos2d-JS中的数据持久化，知道了在Web端数据持久化依赖于`windows.localStorage`，而在Native上数据持久化底层实现依赖于SQLite3数据库。在HTML5上，`windows.localStorage`保存的数据具有安全隐患，所以，除非是一些无关紧要的数据，否则我们都应当将数据放在服务端。除此之外，我们还知道了与数据存储相关的plist和JSON。plist在做配置上，有非常明显的优势，JSON则更多地作为网络传输协议。最后在本章的实例中，通过`localStorage`实现了“天天向上”的解锁功能。

8.7 参考资源

本章的参考资源如下。

- Plist 维基百科：https://zh.wikipedia.org/wiki/Plist。
- JSON ：http://json.org/json-zh.html。
- LocalStorage： https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage。

第 9 章 粒子系统

在游戏中，要表现一些动画效果，例如奔跑的小人、刀光效果等，一般可以通过帧动画或者骨骼动画的方式来实现，然而要表现火焰、烟花、雨雪、水面、烟雾等效果时，虽然采用动画也可以实现，但总是有些差强人意。一方面，帧动画需要美术一帧一帧的绘画，这不仅提高了项目的开发成本，还增加了游戏安装包的大小以及运行时内存的开销。另一方面，帧动画的表现都是预先设定好的，这使得动画的随机性大大降低了，效果的真实性大打折扣。粒子系统就是为了解决此类问题而诞生的，它可以模拟自然界粒子的运动状态，让游戏的效果更加逼真。

本章内容：

- 粒子系统概述
- Cocos2d-JS中的粒子系统
- ParticleDesigner编辑器
- ParticleEditor编辑器
- 在Cocos2d-JS中使用粒子系统

9.1 粒子系统概述

粒子系统是计算机图形学中的一种模拟技术，通常用来模拟一些其他传统的渲染方式难以实现的效果，例如火焰、爆炸、烟雾、水流、落叶、云、雾、雨、雪、尘等。

粒子系统是一个群组的概念，它由大量的粒子组成，每个粒子都有自己的速度、加速度、大小、颜色、生命周期以及纹理等属性，粒子的纹理图片大小一般为64×64像素，图片过大，则会影响渲染性能。

此外，粒子系统中还存在一个粒子发射器，它将粒子系统中所有的粒子以一定的规则发射出去。因为所有的粒子都有自己的大小、速度等属性，这些属性值的不同使得整个粒子产生了随机性，也构成了每个粒子的表现形式以及运动方式，整个粒子系统也就由这些粒子组成。每个粒子系统都是一个整体，例如一团火、一片云、一股烟都是一个独立的粒子系统。

9.2　Cocos2d-JS中的粒子系统

在Cocos2d-JS中，粒子系统相关的类有粒子系统类cc.ParticleSystem、粒子系统批处理类cc.ParticleBatchNode、引擎自带的一些粒子系统示例类，例如cc.ParticleFire、cc.ParticleFireworks等，这些示例都定义在frameworks/cocos2d-html5/cocos2d/particle/CCParticleExamples.js文件中。

说明　Cocos2d-JS中的粒子系统虽然在大部分平台下都支持得很好，但是在Canvas渲染模式下，显得比较吃力，会造成卡顿，这是因为Canvas不支持批处理渲染（参见第11章）。
关于引擎自带的示例粒子系统，读者若有兴趣，可自行查阅CCParticleExamples.js文件中的源码，这里我不打算讲述。因为在实际开发中，很少手写粒子系统属性配置，大部分都是采用粒子系统编辑器的方式实现。

---

cc.ParticleSystem类

cc.ParticleSystem作为粒子系统类，它继承自cc.Node，继承了cc.Node所有的属性和函数。除此之外，cc.ParticleSystem自身定义了众多的属性，这些属性加起来大约有四五十个。虽然属性众多，但是可以将它们分为6大组，然后逐个击破，或许这是一个不错的掌握方法。

根据ParticleDesigner（见9.3节）编辑器的页面布局，将粒子系统的属性进行分组，具体如下：

- 粒子配置（Particle Configuration）
- 发射器类型（Emitter Type）
- 重力配置（Gravity Configuration）
- 径向配置（Radial Configuration）
- 发射器位置（Emitter Location）
- 粒子纹理（Particle Texture）
- 粒子颜色（Particle Color）
- 混合模式（Blend Function）

1. 粒子配置

粒子配置主要就是粒子的基本属性，如生命周期、大小、方向等，见表9-1。

表9-1　粒子系统基础配置属性表

类　型	属　性	说　明
整型	_totalParticles	最大允许粒子数量
整型	particleCount	当前粒子数量

（续）

类　型	属　性	说　明
浮点型	life	粒子生命周期值（秒）
浮点型	lifeVar	粒子生命周期浮动值（秒）
整型	startSize	粒子出生大小
整型	startSizeVar	粒子出生大小浮动值
整型	endSize	粒子消亡时大小
整型	endSizeVar	粒子消亡时大小浮动值
整型	angle	粒子发出方向角度
整型	angleVar	粒子发出方向角度浮动值
整型	startSpin	粒子出生时旋转角度
整型	startSpinVar	粒子出生时旋转角度浮动值
整型	endSpin	粒子消亡时旋转角度
整型	endSpinVar	粒子消亡时旋转角度浮动值
布尔	autoRemoveOnFinish	粒子发射完毕之后是否从父节点移除自身

需要说明的是，当particleCount大于_totalParticles时，粒子系统不再添加粒子，当有粒子死亡时，particleCount会减一，从而又满足了particleCount小于_totalParticles的条件，整个粒子系统中所有粒子的生成就是围绕着这个循环运作。

说明　每种属性的浮动值实际上都是给当前属性产生一个随机值。例如，某一个属性的浮动值是5，那么产生随机数的取值范围就是[0.0, 5.0]。注意，这个区间产生的值是一个浮点型。

2. 发射器类型

与发射器相关的属性有两个，一为粒子发射器模式，二是发射器持续时间，见表9-2。

表9-2　发射器类型相关属性

类　型	属　性	说　明
浮点型	duration	发射持续时间
枚举类型	emitterMode	发射器类型

发射器的持续时间表示多少秒内会持续发射出粒子，其值为浮点型，若duration被设置为–1，则表示一直循环发射粒子。

粒子发射器的类型分为重力（cc.ParticleSystem.MODE_GRAVITY）和半径（cc.ParticleSystem.MODE_RADIUS）模式，在重力模式的情况下，每个粒子可以拥有物理属性，其特点是让粒子达到自然物理效果。

3. 重力配置

在重力模式下，其配置信息如表9-3所示，主要就是重力值和速度相关。

表9-3　重力配置相关属性

类　　型	属　　性	说　　明
cc.p	gravity	重力值
浮点型	speed	粒子的运动速度
浮点型	speedVar	粒子运动速度的浮动值
浮点型	tangentialAccel	切向速度
浮点型	tangentialAccelVar	切向速度浮动值
浮点型	radialAccel	径向速度
浮点型	radialAccelVar	径向速度浮动值

4. 径向配置

径向模式下的配置和重力模式下的配置差不多，其属性见表9-4。

表9-4　径向配置相关属性

类　　型	属　　性	说　　明
浮点型	startRadius	开始半径
浮点型	startRadiusVar	开始半径浮动值
浮点型	endRadius	最小半径
浮点型	endRadiusVar	最小半径浮动值
浮点型	rotatePerSecond	粒子围绕中心点旋转的加速度
浮点型	rotatePerSecondVar	粒子围绕中心点旋转的加速度浮动值

值得说明的是，通过粒子系统编辑器编辑出来的粒子，并没有给endRadiusVar属性配置值，引擎默认将它设置为0。

5. 发射器位置

发射器的位置，实际上可以理解为每个粒子出生时的位置，相关属性见表9-5。

表9-5　发射器位置相关属性

类　　型	属　　性	说　　明
cc.p	_position	发射点坐标
cc.p	_posVar	发射点坐标浮动值

6. 粒子纹理

每个粒子都关联一张纹理图片，图片的大小尺寸一般建议在64×64像素，图片尺寸过大，则会影响渲染性能。其属性如表9-6所示。

表9-6　粒子纹理

类　　型	属　　性	说　　明
cc.Texture2D	_texture	粒子的纹理

7. 粒子颜色

粒子颜色用于设置粒子在整个生命周期里面的颜色变化，其中包含粒子出生时的颜色和到死亡时的颜色。在粒子的生命周期内，粒子的颜色由出生时的颜色渐变到死亡时的颜色，属性如表9-7所示。

表9-7 粒子颜色相关属性

类　型	属　性	说　明
cc.Color	_startColor	粒子出生时的颜色
cc.Color	_startColorVar	粒子出生时的颜色浮动值
cc.Color	_endColor	粒子死亡时的颜色
cc.Color	_endColorVar	粒子死亡时的颜色浮动值

8. 混合函数

图元（精灵和粒子等）都是逐一绘制在帧缓冲（FrameBuffer）中，并最终呈现在屏幕上的。正因为绘制的顺序有先后，所以帧缓冲中任意一个位置的像素颜色在绘制的过程中都会不断地受到两个重要因素的影响：一是当前已经写入帧缓冲中的原有颜色（目标颜色），二是下一个图元期望提交到该像素位置的新颜色（源色）。

混合函数的作用就是围绕用户指定的目标颜色与源颜色进行一些简单的运算，来最终决定写入帧缓冲中该像素的新颜色。一些常见的效果如半透明、正片叠底等，都可以通过制定合适的混合函数来实现。

9

我们可以通过数学公式来表达一下此过程最简单的运算方式（新版的OpenGL支持对混合函数进行更细致的配置）。现假设源颜色的4个分量（红色、绿色、蓝色、alpha值）为(Rs, Gs, Bs, As)，目标颜色的4个分量为(Rd, Gd, Bd, Ad)，又假设上述所说的两个颜色分别有一个乘积系数，其中源因子为(Sr, Sg, Sb, Sa)，目标因子为(Dr, Dg, Db, Da)，则混合产生的新颜色可以表示为：$(Rs \times Sr + Rd \times Dr, Gs \times Sg + Gd \times Dg, Bs \times Sb + Bd \times Db, As \times Sa + Ad \times Da)$。值得说明的是，如果颜色或其对应的系数的分量被截取为 [0.0, 1.0]，对应整数表示的范围为[0, 255]。

在OpenGL中，源因子和目标因子可以通过`glBlendFunc(src, dst)`函数（WebGL下对应`blendFunc(src, dst)`函数）来设置。`glBlendFunc`有两个参数，前者表示源因子，后者表示目标因子。这两个参数可以是多种值，表9-8列出了所有可选值及其说明和计算公式（其中`as`、`ad`为Alpha Source和Alpha Destination的缩写，分别表示源Alpha通道值和目标Alpha通道值。同理`ca`为Constant Alpha的缩写，`cr`、`cg`、`cb`也同理）。

表9-8　**glBlendFunc**函数参数可选值

可　选　值	说　　明
cc.ONE	说明：1.0作为因子 公式：(1, 1, 1, 1)
cc.ZERO	说明：0.0作为因子 公式：(0, 0, 0, 0)
cc.SRC_ALPHA	说明：源色的Alpha值作为因子 公式：(a, a, a, a)
cc.SRC_ALPHA_SATURATE	说明：源色的Alpha值与1减去目标颜色的Alpha值这两者的最小值作为因子 公式：(min(as, 1 - ad), min(as, 1 - ad), min(as, 1 - ad), min(as, 1 - ad))
cc.SRC_COLOR	说明：源色的颜色作为因子 公式：(r, g, b, a)
cc.DST_ALPHA	说明：目标色的Alpha值作为因子 公式：(a, a, a, a)
cc.DST_COLOR	说明：目标色的颜色作为因子 公式：(r, g, b, a)
cc.ONE_MINUS_SRC_ALPHA	说明：1减去源色的Alpha值作为因子 公式：(1 - a, 1 - a, 1 - a, 1 - a)
cc.ONE_MINUS_SRC_COLOR	说明：1减去源色的颜色值作为因子 公式：(1 - r, 1 - g, 1 - b, 1 - a)
cc.ONE_MINUS_DST_ALPHA	说明：1减去目标色的Alpha值作为因子 公式：(1 - a, 1 - a, 1 - a, 1 - a)
cc.ONE_MINUS_DST_COLOR	说明：1减去目标色的颜色值作为因子 公式：(1 - r, 1 - g, 1 - b, 1 - a)
cc.ONE_MINUS_CONSTANT_ALPHA	说明：1减去一个指定的常量Alpha值作为因子 公式：(1 - ca, 1- ca, 1- ca,1 - ca)
cc.ONE_MINUS_CONSTANT_COLOR	说明：1减去一个指定的常量颜色值作为因子 公式：(1 - cr, 1 - cg, 1 - cb, 1 - ca)

粒子系统的_blendFunc属性用来保存源因子和目标因子，如表9-9所示。

表9-9　粒子系统的**_blendFunc**属性

类　　型	属　　性	说　　明
Object	_blendFunc	混合函数需要用到的因子配置

其中，_blendFunc的初始值如下：

```
1  this._blendFunc = {
2      src : cc.BLEND_SRC,      // 源因子
3      dst : cc.BLEND_DST       // 目标因子
4  };
```

需要指明一点，混合函数并不只服务于粒子系统，它是属于渲染层的东西，所以在其他可视节点上依旧可用。

说明　Canvas不支持混合函数。

9.3　ParticleDesigner 编辑器

前面提到过，在一个粒子系统中，有四五十个属性，如果通过代码的方式实现，是一件非常痛苦的事情。一方面，一个成功的粒子往往都是精雕细磨，不断调整参数之后才得到的。另一方面，如果通过代码的方式实现，那么每次调整一个属性，都需要再次运行一下项目，才可以看到效果，这无疑是一种笨拙且费时的方法，间接增加了项目的开发成本。在实际开发中，我们一般通过编辑器的方式去调出理想的粒子系统。

ParticleDesigner是一款非常出色的商业粒子系统编辑器，它不仅提供了粒子模拟器，实现了“所见即所得”，而且还提供了大量的示例，但是较为遗憾的是，ParticleDesigner编辑器仅支持Mac OS系统，其下载地址是：https://71squared.com/particledesigner 。ParticleDesigner的软件页面如图 9-1、图9-2和图9-3所示。



图9-1　粒子效果预览窗口

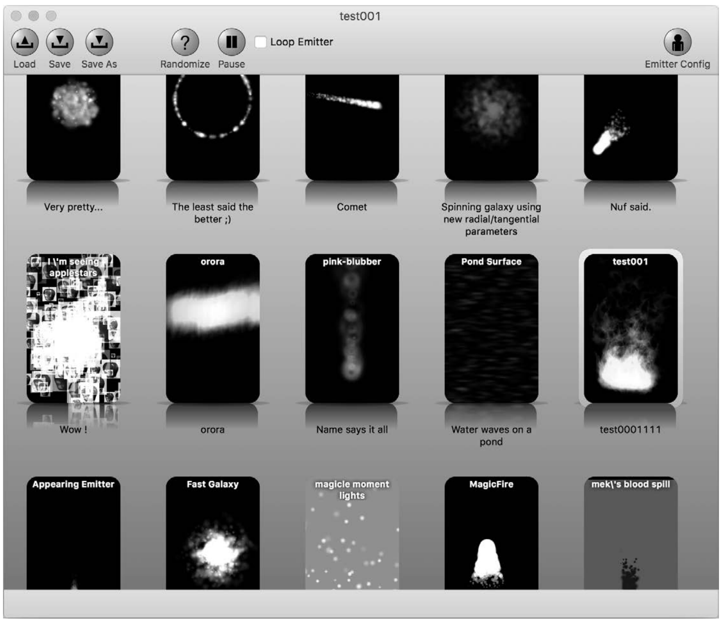


图9-2　ParticleDesigner自带的粒子示例

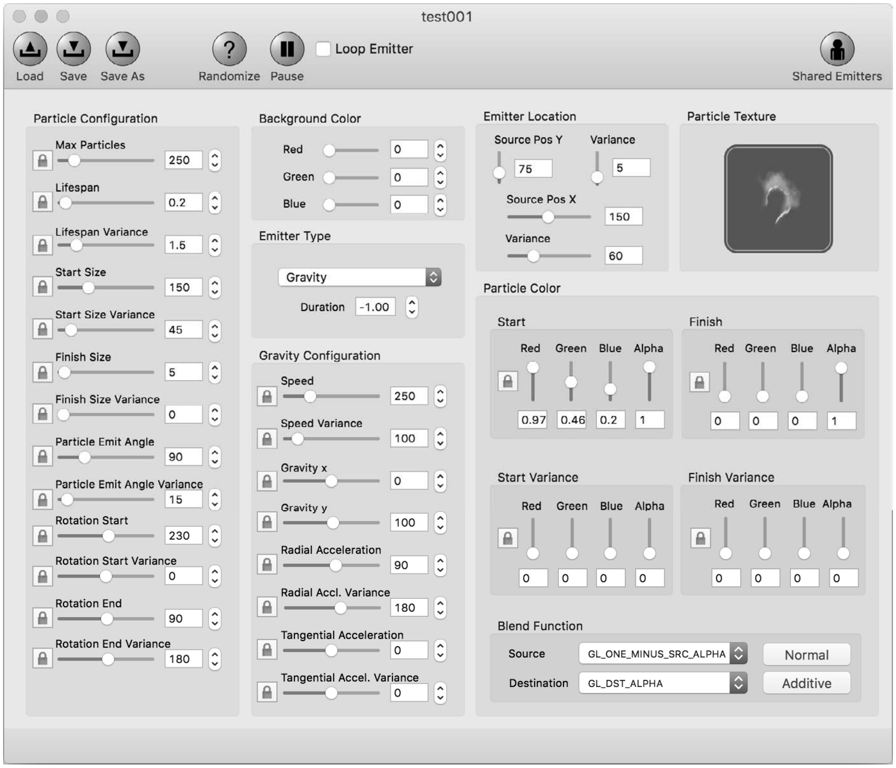


图9-3　粒子属性调整面板

图9-1是粒子效果预览窗口，可以最直观地看到调出来的粒子长什么样。在预览窗口上方有3个按钮，从左往右分别是Orientation、iPad/iPhone和Clear Background。Orientation用于调整屏幕方向，iPad/iPhone用于切换iPhone或者iPad预览窗口，Clear Background用于清除背景颜色。

图9-2为ParticleDesigner自带的粒子示例，图 9-3 是粒子属性配置面板，可以通过修改这些属性来调整出你想要的粒子效果。在图9-2和图 9-3 顶部的工具栏中，都有7个按钮，分别是Load、Save、Save as、Randomize、Pause、Loop Emitter和Emitter Config/Shared Emitters，下面简要介绍一下它们。

- Load：用于加载先前编辑好的粒子文件。
- Save：用于保存当前修改。
- Save as：另存为。
- Randomize：将粒子各个属性随机化，即随机生成一个粒子系统。
- Pause：暂停粒子模拟效果。
- Loop Emitter：一个复选框，用于控制粒子预览是否循环发射。
- Emitter Config/Shared Emitters：用于在图 9-2和图 9-3之间进行切换。

需要说明的是，编辑器上的粒子属性、编辑器生成的plist文件以及`cc.ParticleSystem`类的属性这三者之间的属性名称并不一样。例如粒子的起始大小，在ParticleDesigner中用Start Size来表示，而在生成的plist文件中则用`startParticleSize`表示，在最后的`cc.ParticleSystem`类中，却是用`startSize`属性表示。你可以在`cc.ParticleSystem`类的`initWithDictionary(dictionary, dirname)`函数中看到从plist文件到`cc.ParticleSystem`类属性的转换过程。

虽然这三者有差异，但是并不需要你过于关注，只需将编辑好的粒子导出，然后添加到Cocos2d-JS中即可，详见9.5节。

9.4 ParticleEditor 编辑器

在Windows上一直都不缺好的开发工具，但是粒子编辑器却没有做得比较好的。就目前而言，Particle Editor算是在Windows上面做得最突出的了，图9-4为Particle Editor的主界面。

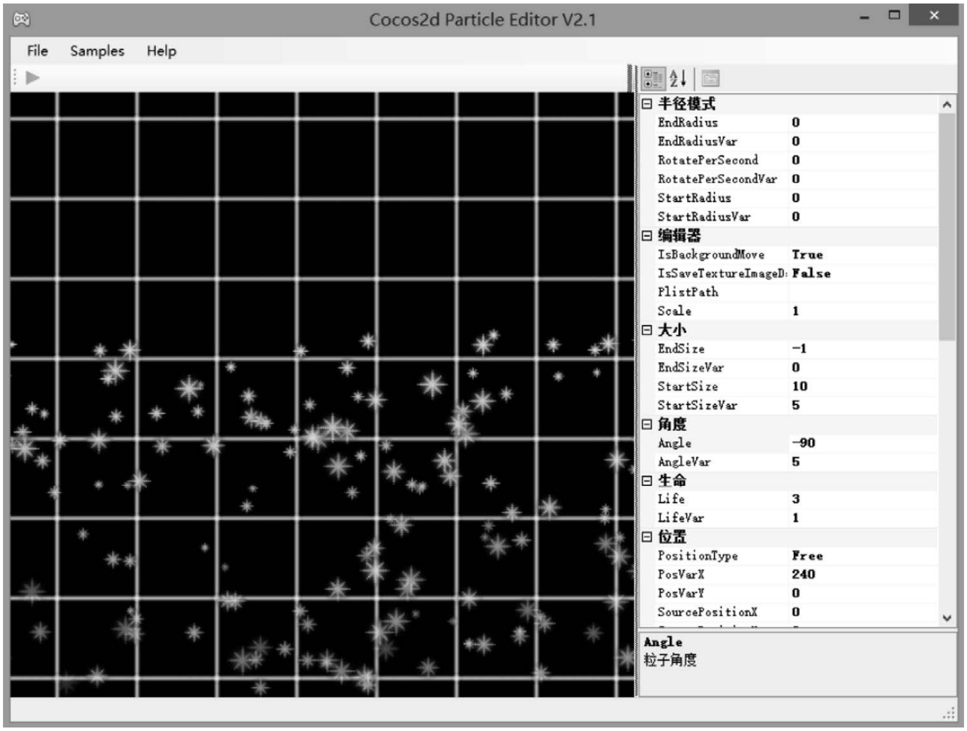


图9-4 Particle Editor的主界面

Particle Editor的下载地址为：https://code.google.com/p/cocos2d-windows-particle-editor/downloads/detail?name=ParticleEditor%20V2.0.7z&can=2&q=（由于此链接较长，所以会在随书代码中贴出）。除此之外，其他的下载途径很多，你可以自行到网上搜索。

Particle Editor的顶部菜单栏有3个菜单项，其中File→New用于新建一个粒子系统模板，File→Open用于加载粒子plist文件，File→Save用于保存当前更改，File→Save as用于将当前文件另存为新文件。Samples菜单项里面主要提供了一些已经编辑好的粒子模板，最后的Help菜单项里只有一个About选项。

下方黑色的网格面板是粒子的预览窗口，预览窗口右边的列表是粒子的属性配置面板，通过它可以调出理想的粒子。

9.5　在 Cocos2d-JS 中使用粒子系统

调整完粒子配置后，可以通过Save as菜单项导出一个plist配置文件。Particle Designer还会导出一张PNG纹理图，如图9-5所示。

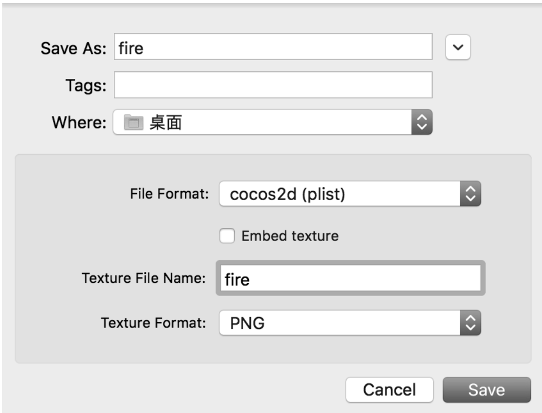


图9-5　使用Particle Designer导出plist格式的粒子文件

在plist文件中，有一个key为textureFileName的键值对，通过修改这个键值对的值可以直接替换掉粒子的纹理贴图。你也可以点击图9-5中的Embed texture复选框，将纹理图片直接保存在plist文件中，这种方式实际上是把纹理图片转为Base64的格式。

将plist配置文件和PNG纹理图片在resource.js文件中声明后，通过如下代码便可以创建出一个粒子系统：

```
1  var particle = new cc.ParticleSystem(res.u9_fire_plist);
2  this.addChild(particle);
3  particle.setPosition(cc.winSize.width/2, cc.winSize.height/2);
```

在上述代码中，第1行代码通过cc.ParticleSystem类来构造一个粒子系统对象，其构造函数接收一个plist文件路径作为参数，cc.ParticleSystem类内部会解析该文件，并创建节点对象。第2行代码的this表示的是当前层，也可以是某一个节点。运行效果如图9-6所示。

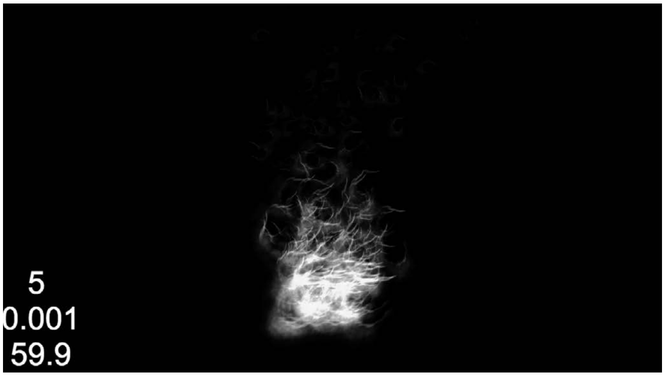


图9-6 火的粒子效果

虽然在Cocos2d-JS游戏引擎中使用粒子系统非常简单方便，但是想在编辑器中调整出一个满意的粒子效果并不简单。在实际开发中，一般会选择在编辑器自带的示例中找到一个和你想要的效果类似的粒子，然后基于这个粒子进行修改。

9.6 小结

通过本章的学习，我们了解了粒子系统的作用及其各个属性配置。学会了在Mac OS X和Windows下使用粒子编辑器，也知道了想要快速调出一个自己满意的粒子效果，可以在ParticleDesigner编辑器自带的示例中找到一个效果和我们想要的效果类似的粒子，在这个粒子上进行调整，可以最快地得到你想要的粒子效果。另外，还知道了Canvas渲染模式下粒子系统的渲染性能支持得不是很好。这是因为Canvas不支持批处理绘制，所以在Canvas中，可以采用简单的帧动画模拟出粒子系统的效果。

9.7 参考资源

本章的参考资源如下。

- ❑ Particle system：https://en.wikipedia.org/wiki/Particle_system。
- ❑ glBlendFunc : https://www.opengl.org/sdk/docs/man/html/glBlendFunc.xhtml。

第10章 UI控件

一款游戏除了核心玩法、AI（Artificial Intelligence，人工智能）之外，还有UI（User Interface，用户界面）等。UI指的是商店、包裹、排行榜等这样的模块。Cocos2d-JS在3.0版本之后，提供了一套较为完善的UI控件，这让我们开发UI模块变得简单许多，这些UI控件的源码位于cocos2d-x/web/extensions/ccui/uiwidgets目录下。

本章内容：

- `ccui.Widget`父类
- `ccui.Text`文本
- `ccui.Button`按钮
- `ccui.CheckBox`复选框
- `ccui.Slider`滑块
- `ccui.ImageView`图片视图
- `ccui.LoadingBar`加载条
- `ccui.TextField`编辑框
- `ccui.Layout`布局
- `ccui.ScrollView`滚动视图
- `ccui.PageView`分页视图
- `ccui.ListView`列表视图
- 实例——《保卫萝卜2》关卡选择页面的开发

10.1 ccui.Widget 父类

在Cocos2d-JS中，UI的命名空间为`ccui`。`ccui.Widget`是所有UI控件的父类，而`ccui.Widget`又间接继承自`cc.Node`，这使得所有的UI控件都具有`cc.Node`的特性；也就是说，每个UI控件都可以有坐标、大小等属性，也可以运行动作等。

既然`ccui.Widget`是所有UI控件的父类，那么我想你非常有必要熟悉一下`ccui.Widget`都有哪些常用的API。这里我将`ccui.Widget`和触摸相关的API提取出来，如表10-1所示。

说明 使用UI控件，需要在project.json的modules数组中添加"ccui"字符串，代码如下：

```
1 "modules" : ["cocos2d", "ccui"],
```

表10-1 ccui.Widget中与触摸相关的API[①]

返回值类型	参数类型	函 数	说 明
无	Boolean	setTouchEnabled(isEnable)	设置是否启用触摸事件
Boolean	无	isTouchEnabled()	判断是否启用触摸事件
无	Function Object	addTouchEventListener(selector, target)	添加触摸事件的回调函数
无	Boolean	setPropagateTouchEvents(isPropagate)	设置触摸事件是否允许向上传播到父节点
Boolean	无	isPropagateTouchEvents()	获取触摸事件是否允许向上传播到父节点的结果
无	Boolean	setSwallowTouches(isSwallow)	设置控件是否吞噬触摸事件
Boolean	无	isSwallowTouches()	判断控件是否吞噬触摸事件
Boolean	cc.Touch cc.Event	onTouchBegan(touch, event)	触摸事件开始时的回调函数
无	cc.Touch cc.Event	onTouchMoved(touch, event)	触摸事件中触点移动时的回调函数
无	cc.Touch cc.Event	onTouchEnded(touch, event)	触摸事件结束时的回调函数
无	cc.Point	onTouchCancelled(touchPoint)	触摸事件取消时的回调函数

从表10-1可以得出，所有的UI控件都支持触摸事件。触摸开关默认是关闭的，你需要通过setTouchEnabled(isEnable)函数启用触摸，并且需要给控件添加一个触摸事件监听器，代码如下：

```
1  node.addTouchEventListener(function(sender, type) {
2      switch (type) {
3          case ccui.Widget.TOUCH_BEGAN:    // 触摸事件开始时响应
4              break;
5          case ccui.Widget.TOUCH_MOVED:    // 触摸事件中触点移动时响应
6              break;
7          case ccui.Widget.TOUCH_ENDED:    // 触摸事件结束时响应
8              break;
9          case ccui.Widget.TOUCH_CANCELED: // 触摸事件取消时响应
10             break;
11         default:
12             break;
```

① 在本章的表中，“参数类型”与“函数”中的参数是一一对应的，即有几个参数，表中就会一一对应地列出多少参数类型来。

```
13     }
14 }.bind(this));
```

可以发现，UI控件的触摸并不需要像给精灵添加触摸事件那样手动判断触点是否在精灵区域内，UI默认的有效触摸区域为自身的大小。另外，在Cocos2d-JS 3.3之后，引擎提供了setSwallowTouches(isSwallow)函数接口，这使得UI控件的触摸也支持事件穿透了。

ccui.Widget还有启用或禁用状态，其API如表10-2所示。

表10-2　ccui.Widget中启用相关的API

返回值类型	参数类型	函　　数	说　　明
无	Boolean	setEnabled(isEnabled)	设置是否启用控件
Boolean	无	isEnabled()	获取控件是否启用的结果

每个UI控件都可以设置启用或者禁用，默认值为true。如果设置为false，则控件是禁用的，不会响应触摸事件。

setBright(isBright)用来设置控件的高亮显示，其相关API如表10-3所示。

表10-3　ccui.Widget中高亮相关的API

返回值类型	参数类型	函　　数	说　　明
无	Boolean	setHighlighted(isHighlight)	设置控件状态为高亮状态
Boolean	无	isHighlighted()	获取控件是否处于高亮状态的结果
无	Boolean	setBright(isBright)	设置控件是否高亮
Boolean	无	isBright()	获取控件是否高亮的结果
无	Number	setBrightStyle(brightStyle)	设置控件的高亮风格

ccui里面的所有控件都有3种状态：normal、highlight和disabled。

bright属性控制UI控件是否变灰，若bright为true，则还需要判断highlight是否为true，如果是true，则是highlight状态，否则是normal状态。所以，如果你想改变一个控件的外观为不可用状态，除了setEnabled(false)以外，还需要调用setBright(false)函数，否则外观不会改变。

另外，brightStyle的可取值如下。

- **ccui.Widget.BRIGHT_STYLE_NORMAL**：表示控件在正常状态。
- **ccui.Widget.BRIGHT_STYLE_HIGH_LIGHT**：表示控件处于高亮状态。

此外，ccui.Widget还提供了焦点操作的相关API，如表10-4所示。焦点就是你当前选中了哪个控件，该控件就获得了焦点。例如按下了某个按钮，或者通过其他的某种方式选中了某个控件，那么此控件就获得了焦点。如果是可编辑的控件，例如编辑框（见10.8节），则会有一个一

闪一闪的竖线在那。如果是不可编辑的控件，一般也会有些高亮显示。

表10-4 ccui.Widget中焦点相关的API

返回值类型	参数类型	函 数	说 明
Boolean	无	isFocused()	查询是否拥有焦点
无	Boolean	setFocused(focus)	切换控件是否拥有焦点
Boolean	无	isFocusEnabled()	查询是否允许接受焦点
无	Boolean	setFocusEnabled(enable)	切换是否允许接受焦点
无	无	requestFocus()	请求获取焦点
无	ccui.Widget ccui.Widget	onFocusChange(lostFocus, getFocus)	焦点更改事件发生时会调用此方法
ccui.Widget	无	getCurrentFocusedWidget()	获取当前拥有焦点的控件

ccui.Widget基于cc.Node之上还扩展了一些和位置有关系的快捷API，例如获取控件左边界在父节点坐标系中的位置，如表10-5所示。

表10-5 ccui.Widget中位置相关的快捷API

返回值类型	参数类型	函 数	说 明
Number	无	getLeftBoundary()	获取左边界在父节点坐标系中的位置
Number	无	getBottomBoundary()	获取下边界在父节点坐标系中的位置
Number	无	getRightBoundary()	获取右边界在父节点坐标系中的位置
Number	无	getTopBoundary()	获取上边界在父节点坐标系中的位置
cc.p	无	getWorldPosition()	获取控件在世界坐标系中的位置

最后，剩下一些零碎的API，了解一下即可，如表10-6所示。

表10-6 ccui.Widget百分比和启用布局组件API

返回值类型	参数类型	函 数	说 明
Number	无	getSizePercent()	获取控件的百分比尺寸
无	Number	setSizePercent(percent)	设置控件的百分比尺寸
Boolean	无	isLayoutComponentEnabled()	获取是否启用布局组件的结果
无	Boolean	setLayoutComponentEnabled(isEnable)	设置是否启用布局组件

10.2 ccui.Text 文本

Text文本用来显示玩家的名字、等级、钻石数量等，它和cc.Label标签类似，也分为TTF、BMFont和Atlas三种类型。

10.2.1 ccui.Text 字体文本

ccui.Text支持TTF字库，创建一个ccui.Text文本的代码如下：

```
1  var node = new ccui.Text("我是Text文本","AmericanTypewriter",30);
```

其中，AmericanTypewriter为系统自带的TTF字体，30为字号大小。ccui.Text的构造函数以及参数说明如表10-7所示。

表10-7　ccui.Text构造函数以及参数说明

构造函数	*ccui*.Text(textContent, fontName, fontSize);	
参数列表	参　数	参数说明
	textContent	文本内容
	fontName	字体文件或字体名称
	fontSize	字体大小

此外，ccui.Text常用函数如表10-8所示。

表10-8　cc.Text常用函数

返回值类型	参数类型	函　数	说　明
String	无	getString()	获取内容
无	String	setString(text)	设置内容
cc.Size	无	getFontSize()	获取字号大小
无	cc.Size	setFontSize(size)	设置字号大小
String	无	getFontName()	获取字体文件或名称
无	String	setFontName(name)	设置字体文件或名称
cc.Color	无	getTextColor()	获取文本颜色
无	cc.Color	setTextColor(color)	设置文本颜色
Number	无	getType()	获取类型
Number	无	getTextHorizontalAlignment()	获取文本水平对齐方式
无	Number	setTextHorizontalAlignment(alignment)	设置文本水平对齐方式
Number	无	getTextVerticalAlignment()	获取文本垂直对齐方式
无	Number	setTextVerticalAlignment(alignment)	设置文本垂直对齐方式
无	cc.Color cc.Size Number	enableShadow(shadowColor, offset, blurRadius)	启用文本阴影效果，仅TTF字体对象有效
无	cc.Color cc.Size	enableOutline(outlineColor, outlineSize)	启用文本描边效果，仅TTF字体对象有效
无	cc.Color	enableGlow(glowColor)	启用文本发光效果，仅TTF字体对象有效

其中，文本类型可取的值如下所示。

- **ccui.Text.Type.SYSTEM**：表示系统字体。
- **ccui.Text.Type.TTF**：表示TTF文件字体。

水平对齐方式可取的值如下所示。

- **cc.TEXT_ALIGNMENT_LEFT**：表示左对齐。
- **cc.TEXT_ALIGNMENT_CENTER**：表示居中对齐。
- **cc.TEXT_ALIGNMENT_RIGHT**：表示右对齐。

垂直对齐方式可取的值如下所示。

- **cc.VERTICAL_TEXT_ALIGNMENT_TOP**：表示上对齐。
- **cc.VERTICAL_TEXT_ALIGNMENT_CENTER**：表示居中对齐。
- **cc.VERTICAL_TEXT_ALIGNMENT_BOTTOM**：表示下对齐。

10.2.2 ccui.TextBMFont 位图文本

ccui.TextBMFont与cc.LabelBMFont类似，它支持FNT类型的文件。创建一个ccui.TextBMFont的代码如下：

```
1  var node = new ccui.TextBMFont("12345", "res/unit10_ui/fonts/number.fnt");
```

其中，12345为设置的文本内容，res/unit10_ui/fonts/number.fnt为指定的FNT字体文件，FNT字体文件的制作见3.7.4节。

10

ccui.TextBMFont的构造函数以及参数说明如表10-9所示。

表10-9 ccui.TextBMFont构造函数以及参数说明

构造函数	ccui.TextBMFont(text, filename)	
参数列表	参 数	参数说明
	text	文本内容
	fontName	字体文件或字体名称

此外，ccui.TextBMFont常用函数如表10-10所示。

表10-10 ccui.TextBMFont常用函数

返回值类型	参数类型	函 数	说 明
String	无	getString()	获取文本内容
无	String	setString(value)	设置文本内容
无	String	setFntFile(fileName)	设置FNT字体文件或者字体名称

10.2.3　ccui.TextAtlas 字符映射文本

ccui.TextAtlas为字符映射文本，与cc.LabelAtlas也是类似的。创建一个ccui.TextAtlas对象的代码如下：

```
1  var node = new ccui.TextAtlas("789", "res/unit10_ui/fonts/score.png", 40, 50, '0');
```

ccui.TextAtlas的构造函数以及参数说明如表10-11所示。

表10-11　ccui.TextAtlas构造函数以及参数说明

构造函数	ccui.TextAtlas(strText, charMapFile, itemWidth, itemHeight, startCharMap)	
参数列表	参　　数	参数说明
	strText	文本内容
	charMapFile	图片名称
	itemWidth	每个“小块”的宽度
	itemHeight	每个“小块”的高度
	startCharMap	起始字符

此外，ccui.TextAtlas常用的函数如表10-12所示。

表10-12　ccui.TextAtlas常用函数

返回值类型	参数类型	函　　数	说　　明
String	无	getString()	获取文本内容
无	String	setString(value)	设置文本内容

10.3　ccui.Button 按钮

按钮（ccui.Button）可以响应回调函数，实现一些逻辑处理。按钮的使用方式也比较简单，创建代码如下：

```
1  // 加载[按钮]
2  loadButton : function(){
3      var node = new ccui.Button();
4      this.addChild(node);
5      node.loadTextures(res.u10_btn_start_normal_png, res.u10_btn_start_pressed_
       png, "");
6      node.setPosition(200, cc.winSize.height / 2);
7      node.setTouchEnabled(true);
8      node.addTouchEventListener(this.onButtonTouchEvent, this);
9  }
```

其中，第5行代码用于设置按钮3个状态的纹理，这3个状态分别是常态、按下和禁用。你也可以将这3个纹理直接放在构造函数中，就像下面这样：

```
1  var node = new ccui.Button(res.u10_btn_start_normal_png, res.u10_btn_start_
   pressed_png, "");
```

ccui.Button的构造函数以及参数说明如表10-13所示。

表10-13 ccui.Button构造函数以及参数说明

构造函数	ccui.Button(normalImage, selectedImage, disableImage, texType)	
参数列表	参　数	参数说明
	normalImage	常态纹理图片
	selectedImage	按下纹理图片
	disableImage	禁用纹理图片
	texType	纹理图片类型

其中texType可取的值如下。

- **ccui.Widget.LOCAL_TEXTURE**：表示本地单张图片（小图）类型。
- **ccui.Widget.PLIST_TEXTURE**：表示精灵表单（大图）类型。

另外，第7行代码设置了按钮可以触摸。实际上，每一个按钮都需要手动开启触摸，并为此按钮添加事件监听器，如第8行代码。onButtonTouchEvent是一个事件回调函数，具体代码如下：

```
1  onButtonTouchEvent : function(sender, type) {
2      switch (type) {
3          case ccui.Widget.TOUCH_BEGAN: // 触摸[按下]
4              cc.log("Touch Down");
5              sender.x += 100; // 按钮坐标右移100
6              break;
7          case ccui.Widget.TOUCH_MOVED: // 触摸[移动]
8              cc.log("Touch Move");
9              break;
10         case ccui.Widget.TOUCH_ENDED: // 触摸[抬起]
11             cc.log("Touch Up");
12             break;
13         case ccui.Widget.TOUCH_CANCELED: // 触摸[取消]
14             cc.log("Touch Cancelled");
15             break;
16         default:
17             break;
18     }
19 }
```

第1行代码中的参数sender为事件源，即按钮自身，按钮被按下之后，按钮将右移100个单位，这在第5行代码中实现。

另外，ccui.Button的3个状态的纹理也可以通过表10-14中的API进行设定。

表10-14　ccui.Button的3个状态纹理的相关函数

返回值类型	参数类型	函　　数	说　　明
无	String String String Number	loadTextures(normal, selected, disabled,texType)	加载按钮的纹理以及纹理类型
无	String Number	loadTextureNormal(normal, texType)	加载按钮正常状态的纹理
无	String Number	loadTexturePressed(selected, texType)	加载按钮选中状态纹理
无	String Number	loadTextureDisabled(disabled, texType)	加载按钮禁用状态的纹理

此外，ccui.Button支持九宫格，你可以通过表10-15中的API设置启用九宫格按钮。启用了九宫格按钮之后，你可以通过setContentSize函数来设置按钮的大小。

表10-15　ccui.Button九宫格相关的函数

返回值类型	参数类型	函　　数	说　　明
Boolean	无	isScale9Enabled()	判断按钮是否为九宫格模式
无	Boolean	setScale9Enabled(enable)	设置按钮为九宫格模式

九宫格的按钮大小可以随意设定，这意味着，按钮中不会有美术做好的字体在上面，你可以通过表10-16中相关的API给按钮添加一些文本内容。

表10-16　ccui.Button文本内容相关函数

返回值类型	参数类型	函　　数	说　　明
String	无	getTitleText()	获取按钮文本内容
无	String	setTitleText(text)	设置按钮文本内容
cc.Color	无	getTitleColor()	获取按钮文本颜色
无	cc.Color	setTitleColor(color)	设置按钮文本颜色
cc.Size	无	getTitleFontSize()	获取按钮文本字号大小
无	cc.Size	setTitleFontSize(size)	设置按钮文本字号大小
String	无	getTitleFontName()	获取按钮文本名字
无	String	setTitleFontName(fontName)	设置按钮文本名字

在实际开发中，你可能遇到的需求是按钮仅有一张纹理支持，它的点击效果是放大和缩小。实际上，ccui.Button在按下的时候就可以有缩放的效果，你可以通过表10-17中的相关API来享受这一效果。

表10-17　ccui.Button缩放效果相关函数

返回值类型	参数类型	函　　数	说　　明
Number	无	getZoomScale()	获取一个缩放比例
无	Number	setZoomScale(scale)	设置一个缩放比例
无	Boolean	setPressedActionEnabled(enabled)	设置启用按钮被按下时的缩放操作

10.4　ccui.CheckBox 复选框

复选框（ccui.CheckBox）有选中和未选中这两个状态，可以为其添加事件监听器来响应事件从而处理对应的逻辑。在游戏中，复选框一般用在一些设置中，例如是否开启特效、是否接受组队邀请、是否接受系统推送等。复选框选中和未选中的状态如图10-1和图10-2所示。



图10-1　CheckBox选中状态



图10-2　CheckBox未选中状态

10

创建一个ccui.CheckBox对象的代码如下：

```
1  // 加载[CheckBox]
2  loadCheckBox : function(){
3      var node = new ccui.CheckBox();
4      this.addChild(node);
5      node.loadTextures("res/unit10_ui/check_box_normal.png",
6          "res/unit10_ui/check_box_normal_press.png",
7          "res/unit10_ui/check_box_active.png",
8          "res/unit10_ui/check_box_normal_disable.png",
9          "res/unit10_ui/check_box_active_disable.png");
10     node.setTouchEnabled(true);
11     node.setPosition(cc.winSize.width/ 2, cc.winSize.height / 2);
12     node.addEventListener(this.onCheckBoxSelectedEvent, this);
13 }
```

第12行为复选框添加了一个事件监听器，其代码如下：

```
1  onCheckBoxSelectedEvent: function (sender, type) {
2      switch (type) {
3          case  ccui.CheckBox.EVENT_UNSELECTED:
4              cc.log("复选框没选中");
5              break;
6          case ccui.CheckBox.EVENT_SELECTED:
7              cc.log("复选框选中");
8              break;
```

```
9           default:
10              break;
11      }
12 }
```

可以看到，ccui.CheckBox可以响应两种类型的事件，分别是未选中ccui.CheckBox.EVENT_UNSELECTED以及选中ccui.CheckBox.EVENT_SELECTED。

另外，ccui.CheckBox的相关纹理读取也可以放在构造函数中，ccui.CheckBox的构造函数以及参数说明如表10-18所示。

表10-18　**ccui.CheckBox**构造函数以及参数说明

构造函数	ccui.CheckBox(backGround, backGroundSelected, cross, backGroundDisabled, frontCrossDisabled, texType)	
参数列表	参　　数	参数说明
	backGround	未选中常态纹理
	backGroundSelected	背景选中状态纹理
	cross	勾选选中状态纹理
	backGroundDisabled	背景禁用状态纹理
	frontCrossDisabled	勾选禁用状态纹理
	texType	纹理类型

此外，ccui.CheckBox常用函数如表10-19所示。

表10-19　**ccui.CheckBox**常用函数

返回值类型	参数类型	函　　数	说　　明
无	String String String String String Number	loadTextures(backGround, backGroundSelected, cross, backGroundDisabled, frontCrossDisabled, texType)	加载相关状态纹理以及纹理类型
无	String Number	loadTextureBackGround(backGround, texType)	加载未选中常态纹理以及纹理类型
无	String Number	loadTextureBackGroundSelected(backGroundSelected, texType)	加载背景选中状态纹理以及纹理类型
无	String Number	loadTextureFrontCross(cross, texType)	加载选中状态纹理以及纹理类型
无	String Number	loadTextureBackGroundDisabled(backGroundDisabled, texType)	加载背景禁用状态纹理以及纹理类型
无	String Number	loadTextureFrontCrossDisabled(frontCrossDisabled, texType)	加载勾选禁用状态纹理以及纹理类型
Boolean	无	isSelected()	获取是否选中的结果
无	Boolean	setSelected(selected)	设置是否选中
无	Function Object	addEventListener(selector, target)	添加事件监听器

10.5 ccui.Slider 滑块

滑块（ccui.Slider）一般用来控制音量或者是调整屏幕暗度等，例如iOS系统中的控制中心，如图10-3所示。



图10-3 iOS系统中的控制中心

滑块控件由滑块控制点、滑块进度条和滑块背景组成，这意味着你需要提供至少3张图片来创建一个Slider滑块控件。创建Slider滑块控件的代码如下：


```
1  // 加载[滑块]
2  loadSlider : function(){
3      var slider = new ccui.Slider();
4      this.addChild(slider);
5      slider.loadBarTexture("res/unit10_ui/slider_bar.png");
6      slider.loadSlidBallTextures(
7          "res/unit10_ui/slider_ball_normal.png",
8          "res/unit10_ui/slider_ball_pressed.png",
9          "res/unit10_ui/slider_ball_disabled.png");
10     slider.loadProgressBarTexture("res/unit10_ui/slider_progress.png");
11     slider.setPosition(cc.winSize.width/2, cc.winSize.height/2);
12     slider.addEventListener(this.onSliderEvent, this);
13 }
```

其中，第5行代码用于加载滑块背景纹理，第6行到第9行代码加载了3张纹理，这3张纹理分别表示滑块控制点的常态纹理、按下纹理和禁用纹理。第10行代码加载了滑块进度条，而第12行代码则给滑块添加一个事件监听器，在此事件监听器的回调函数中，你可以监听到滑块的百分比值，代码如下：

```
1  onSliderEvent: function (sender, type) {
2      switch (type) {
3          case ccui.Slider.EVENT_PERCENT_CHANGED:
```

```
4               var percent = sender.getPercent();
5               cc.log("百分比：  " + percent.toFixed(0)); // 取整输出
6               break;
7           default:
8               break;
9       }
10 }
```

运行上述代码，将滑块的控制点拖到如图10-4所示的位置，此时在Chrome控制台的输出如图10-5所示。



图10-4　Slider滑块

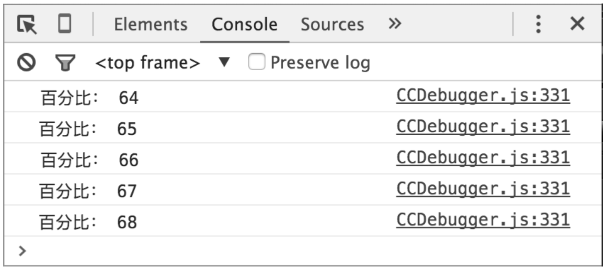


图10-5　Chrome控制台输出

10.6　ccui.ImageView 图片视图

图片视图（ccui.ImageView）实际上就是一张单纯的图片，一般作为UI面板的背景图等，例如图10-6中的木板背景。



图10-6　天天向上解锁对话框

ccui.ImageView的使用是所有UI控件中最简单的，其创建代码如下：

```
1  // 加载[ImageView]
2  loadImageView : function(){
```

```
3       var node = new ccui.ImageView("res/node_256.png");
4       this.addChild(node);
5       node.setPosition(cc.winSize.width/2, cc.winSize.height / 2 );
6   }
```

此外，ccui.ImageView常用的函数如表10-20所示。

表10-20 ccui.ImageView常用函数

返回值类型	参数类型	函　数	说　明
无	String Number	loadTexture(fileName, texType)	加载纹理图片以及纹理图片类型
无	cc.Rect	setTextureRect(rect)	设置纹理区域
无	Boolean	setScale9Enabled(enable)	设置为九宫格
Boolean	无	isScale9Enabled()	获取是否为九宫格的结果

其中，texType与ccui.Button构造函数的参数一致，为纹理图片类型，具体见表10-7。

10.7 ccui.LoadingBar 加载条

加载条（ccui.LoadingBar）也称为进度条，一般用来显示资源加载进度、网络资源下载进度或者怪物的血条等。

创建一个ccui.LoadingBar，仅需要一张图片即可，其代码如下：

```
1   // 加载[进度条]
2   loadLoadingBar : function(){
3       var node = new ccui.LoadingBar();
4       this.addChild(node);
5       node.loadTexture("res/unit10_ui/slider_progress.png");
6       node.setDirection(ccui.LoadingBar.TYPE_LEFT); // 设置方向
7       node.setPercent(10); // 设置百分比
8       node.setPosition(cc.winSize.width/2, cc.winSize.height / 2 );
9   }
```

第6行代码设置LoadingBar的方向为向左，其值还可以是ccui.LoadingBar.TYPE_RIGHT，表示向右。另外，第7行代码用于设置LoadingBar的百分比。实际上，你可以通过一个调度器来动态设置LoadingBar的百分比，从而达到进度条加载的效果，如图10-7所示。

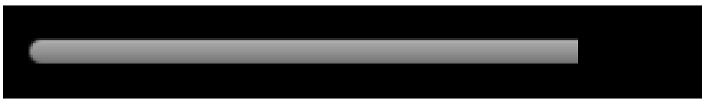


图10-7 LoadingBar加载条

此外，ccui.LoadingBar的常用函数与ccui.ImageView的常用函数是一样的，见表10-20。

10.8　ccui.TextField 编辑框

编辑框（ccui.TextField）一般用来收集一些用户输入的信息，例如玩家注册一个账号时，编辑框可以获取到玩家输入的账号。另外，编辑框也可以设为密码模式，在密码模式下，玩家输入的字符将由设定字符显示（例如*号），而非明文。

创建一个编辑框的代码如下：

```
1  // 加载[编辑框]
2  loadTextFiled : function(){
3      var node = new ccui.TextField("请输入账号：", "Arial", 30);
4      this.addChild(node);
5      node.setPosition(cc.winSize.width/2, cc.winSize.height / 2 );
6      node.addEventListener(this.onTextFieldEvent, this); // 添加事件监听器
7  },
8  // 事件[编辑框输入事件]
9  onTextFieldEvent: function(textField, type){
10     switch (type) {
11         case ccui.TextField.EVENT_ATTACH_WITH_IME:
12             cc.log("[挂载到]输入法编辑器");
13             break;
14         case ccui.TextField.EVENT_DETACH_WITH_IME:
15             cc.log("输入法编辑器[失去挂载]");
16             break;
17         case ccui.TextField.EVENT_INSERT_TEXT:
18             cc.log("输入法编辑器[输入]");
19             break;
20         case ccui.TextField.EVENT_DELETE_BACKWARD:
21             cc.log("输入法编辑器[删除]");
22             break;
23         default:
24             break;
25     }
26     cc.log("编辑框中内容：", textField.getString());
27 }
```

在实际开发中，你可以在case ccui.TextField.EVENT_ATTACH_WITH_IME中写一个Action让textField运行一个向上移动的moveTo动作，然后在case ccui.TextField.EVENT_DETACH_WITH_IME里让textField输入框moveTo回来，这便解决了在手机上键盘可能挡住textField控件的问题。

前面说到，ccui.TextField还有个密码模式，密码模式的开启以及密码显示的样式设定实现代码如下：

```
1  node.setPasswordEnabled(true);  // 启用[密码模式]
2  node.setPasswordStyleText("*"); // 设置[密码样式]
```

此外，ccui.TextField常用函数如表10-21所示。

表10-21 **ccui.TextField**常用函数

返回值类型	参数类型	函 数	说 明
无	ccui.TextField	onTextFieldAttachWithIME(sender)	挂载输入法时的回调函数
无	ccui.TextField	onTextFieldDetachWithIME(sender)	取消挂载输入法时的回调函数
无	ccui.TextField String Number	onTextFieldInsertText (sender, text, len)	插入文本时的回调函数，其参数如下。 ❑ sender：文本编辑框实例。 ❑ text：将要插入的字符串。 ❑ len：将要插入的字符串长度
无	ccui.TextField String Number	onTextFieldDeleteBackward (sender, delText, len)	删除文本时的回调函数，其参数如下。 ❑ sender：文本编辑框实例。 ❑ delText：将要删除的字符串。 ❑ len：将要插入的字符串长度
无	String Number	insertText(text, len)	插入文本
无	无	deleteBackward()	回退删除
无	无	openIME()	打开输入法编辑器
无	无	closeIME()	关闭输入法编辑器
无	Boolean	setMaxLengthEnabled(isEnable)	启用最大长度限制
Boolean	无	isMaxLengthEnabled()	获取是否启用了最大长度限制的结果
无	Number	setMaxLength(length)	设置最大长度
Number	无	getMaxLength()	获取最大长度
无	Number	getCharCount()	获取总共输入的字符数
无	Boolean	setPasswordEnabled(isEnable)	启用密码输入模式
Boolean	无	isPasswordEnabled()	获取是否启用密码输入模式的结果
无	String	setPasswordStyleText(styleText)	设置密码模式下的文本样式
无	String	setPasswordText(text)	设置密码
无	Boolean	setAttachWithIME(isAttach)	设置是否挂载输入法编辑器
无	Boolean	getAttachWithIME()	获取是否挂载输入法编辑器的结果
无	Boolean	setDetachWithIME(isDetach)	设置是否取消挂载输入法编辑器
Boolean	无	getDetachWithIME()	获取是否取消挂载输入法编辑器的结果
无	Boolean	setInsertText(insert)	设置是否允许文本输入
Boolean	无	getInsertText()	获取是否允许文本输入的结果
无	Boolean	setDeleteBackward (isDeleteBackward)	设置是否允许回删
Boolean	无	getDeleteBackward()	获取是否允许回删的结果

10.9　ccui.Layout 布局

布局（ccui.Layout）是一个包含控件的容器，容器中的子节点根据布局类型进行摆放。布局类型有绝对布局、相对布局、水平布局和垂直布局4种，默认为绝对布局。

ccui.Layout和cc.Layer类似，是一个默认看不见的节点。不过，你可以给ccui.Layout设置颜色和内容大小，那么整个ccui.Layout就很直观地显示出来，这是一个非常好的辅助功能。另外，除了设置颜色之外，ccui.Layout还允许设置背景图像。通过如下代码可以创建出一个简单的ccui.Layout：

```
1  // 加载[默认布局]
2  loadLayout : function() {
3      var layout = new ccui.Layout();
4      this.addChild(layout);
5      layout.setContentSize(852, 480);
6      layout.x = (cc.winSize.width - layout.width ) / 2;
7      layout.y = (cc.winSize.height - layout.height) / 2;
8      layout.setBackGroundColor(cc.color(128, 128, 128));
9      // 设置背景颜色类型
10     layout.setBackGroundColorType(ccui.Layout.BG_COLOR_SOLID);
11 }
```

ccui.Layout的默认锚点为cc.p(0, 0)。所以，第6行和第7行代码是将layout定位在屏幕的正中心。第8行代码设置了layout的背景颜色，第9行代码设置了背景颜色的类型。ccui.Layout中背景颜色、背景图片相关的API分别如表10-22和表10-23所示。

表10-22　ccui.Layout中背景颜色相关的API

参数或返回类型	函　　数	说　　明
Number	getBackGroundColorType()	获取背景颜色类型
Number	setBackGroundColorType(type)	设置背景颜色类型
cc.Color	getBackGroundColor()	获取背景颜色
cc.Color	setBackGroundColor(color, endColor)	设置背景颜色
cc.Color	getBackGroundStartColor()	获取渐变背景颜色开始的值
cc.Color	getBackGroundEndColor()	获取渐变背景颜色结束的值
Number	getBackGroundColorOpacity()	获取背景颜色不透明度
Number	setBackGroundColorOpacity(opacity)	设置背景颜色不透明度
cc.p	getBackGroundColorVector()	获取背景颜色向量
cc.p	setBackGroundColorVector(vector)	设置背景颜色向量

其中背景颜色类型可取的值如下。

- **ccui.Layout.BG_COLOR_NONE**：表示无颜色类型。
- **ccui.Layout.BG_COLOR_SOLID**：表示固定颜色类型。

- **ccui.Layout.BG_COLOR_GRADIENT**：表示渐变颜色类型。

表10-23 ccui.Layout与背景图片相关的API

返回值类型	返回类型	函　　数	说　　明
无	String Number	setBackGroundImage(fileName, texType)	设置背景图片
无	无	removeBackGroundImage()	移除背景图片
cc.Color	无	getBackGroundImageColor()	获取背景图片颜色
无	cc.Color	setBackGroundImageColor(color)	设置背景图片颜色
Number	无	getBackGroundImageOpacity()	获取背景图片不透明度
无	Number	setBackGroundImageOpacity(opacity)	设置背景图片不透明度
cc.Size	无	getBackGroundImageTextureSize()	获取背景图片纹理大小
Boolean	无	isBackGroundImageScale9Enabled()	背景图片是否启用九宫格
无	Boolean	setBackGroundImageScale9Enabled(enable)	开启/关闭背景图片九宫格模式

其中，背景图片的纹理类型（texType）可选值如下。

- **ccui.Widget.LOCAL_TEXTURE**：表示单张纹理图片类型。
- **ccui.Widget.PLIST_TEXTURE**：表示精灵表单图集类型。

当创建好一个ccui.Layout之后，便可以往此布局中添加子控件，然后进行布局。ccui.LayoutParameter是一个布局参数类，用来设置布局的排列方式，例如左对齐、右对齐或者水平居中等。ccui.LayoutParameter有两个子类——线性布局类ccui.LinearLayoutParameter和相对布局类ccui.RelativeLayoutParameter。我将layout的布局设置为线性布局，然后给layout添加了3个按钮，代码如下：

```
1  // 加载[垂直布局]
2  loadLayoutLinearVertical : function(){
3      // ......
4      layout.setLayoutType(ccui.Layout.LINEAR_VERTICAL);
5      var buttonTexture = "res/unit10_ui/btn_start_normal.png";
6      // 按钮1
7      var button1= new ccui.Button();
8      layout.addChild(button1);
9      button1.setTouchEnabled(true);
10     button1.loadTextures(buttonTexture, buttonTexture, "");
11     // 布局参数
12     var lp1 = new ccui.LinearLayoutParameter();
13     button1.setLayoutParameter(lp1);
14     lp1.setGravity(ccui.LinearLayoutParameter.CENTER_HORIZONTAL);
15     lp1.setMargin(new ccui.Margin(0, 10, 0, 10));
16     // 按钮2 ......
17     // 按钮3 ......
18 }
```

第4行代码设置了layout的布局类型，其可选值如下。

- **ccui.Layout.ABSOLUTE**：绝对布局类型。
- **ccui.Layout.RELATIVE**：相对布局类型。
- **ccui.Layout.LINEAR_VERTICAL**：线性垂直布局类型。
- **ccui.Layout.LINEAR_HORIZONTAL**：线性水平布局类型。

第12行创建了线性布局参数。第13行给按钮设置了布局参数，注意这里是给按钮而不是给layout设置。

第14行给布局参数对象设置了对齐方式，其可选值如下。

- **ccui.LinearLayoutParameter.NONE**：无对齐方式。
- **ccui.LinearLayoutParameter.LEFT**：左对齐。
- **ccui.LinearLayoutParameter.TOP**：上对齐。
- **ccui.LinearLayoutParameter.RIGHT**：右对齐。
- **ccui.LinearLayoutParameter.BOTTOM**：下对齐。
- **ccui.LinearLayoutParameter.CENTER_VERTICAL**：垂直居中对齐。
- **ccui.LinearLayoutParameter.CENTER_HORIZONTAL**：水平居中对齐。

第15行代码给布局参数对象设置了外边距，外边距的4个参数分别表示的是左边距、上边距、右边距、下边距。

另外，还有两个按钮的开发与按钮1类似（见随书源码）。这里需要说明的是，我并没有给任何一个按钮设置坐标，但是运行上面的代码，可以看到效果如图10-8所示，这就是布局的作用。

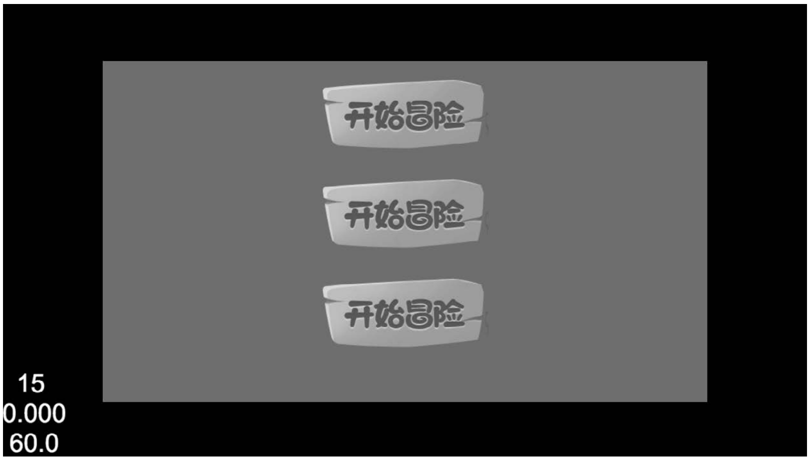


图10-8　垂直居中布局

最后，ccui.Layout剩下的相关API如表10-24所示。

表10-24 **ccui.Layout**其他API

返回值类型	参数类型	函 数	说 明
Number	无	getLayoutType()	获取布局类型
无	Number	setLayoutType(type)	设置布局类型
无	无	requestDoLayout()	请求刷新布局
无	无	forceDoLayout()	强制刷新布局
Boolean	无	isClippingEnabled()	获取是否开启了裁剪的结果
无	Boolean	setClippingEnabled(enable)	设置是否开启裁剪
Boolean	无	isLoopFocus()	获取是否启用循环焦点的结果
无	Boolean	setLoopFocus(loop)	设置是否启用循环焦点
Boolean	无	isPassFocusToChild()	获取是否允许传递焦点到子节点上的结果
无	Boolean	setPassFocusToChild(pass)	设置是否允许传递焦点到子节点上
Number	无	getClippingType()	获取裁剪类型
无	Number	setClippingType(type)	设置裁剪类型
无	Number ccui.Widget ccui.Widget	findNextFocusedWidget(direction, current)	获取指定方向中的下一个能够具有焦点的控件

findNextFocusedWidget(direction, current)中的current表示当前焦点控件，而direction表示方向，其可选值如下。

- **ccui.Widget.LEFT**：向左。
- **ccui.Widget.RIGHT**：向右。
- **ccui.Widget.UP**：向上。
- **ccui.Widget.DOWN**：向下。

另外，ccui.Layout是可以裁剪的。调用setClippingEnabled(true)函数之后，再结合setContentSize(size)即可。下面我给前面的layout开启了裁剪：

```
1  // 加载[垂直布局]
2  loadLayoutLinearVertical : function(){
3      // ......
4      layout.setClippingEnabled(true);
5      layout.setContentSize(430, 200);
6      // ......
7  }
```

运行代码，效果如图10-9所示。

10

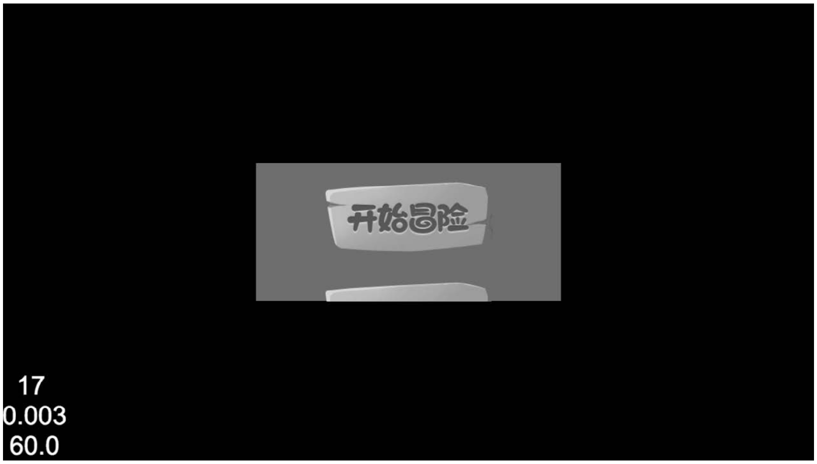


图10-9　ccui.Layout开启裁剪

10.10　ccui.ScrollView 滚动视图

滚动视图（ccui.ScrollView）继承自ccui.Layout，也是一个布局类，它可以让放置在其内部的UI控件滚动显示。例如，《保卫萝卜2》中关卡选择场景中有十几张背景图片，这些图片展开会超出包裹区域，此时便可以使用ccui.ScrollView来布局，使其滚动显示。

创建一个ccui.ScrollView的代码如下：

```
1  // 加载[滚动视图]
2  loadScrollView : function(){
3      var scrollView = new ccui.ScrollView();
4      this.addChild(scrollView);
5      scrollView.setContentSize(cc.size(512, 400));
6      scrollView.x = (cc.winSize.width - scrollView.width ) / 2;
7      scrollView.y = (cc.winSize.height - scrollView.height ) / 2;
8      scrollView.setTouchEnabled(true);
9      scrollView.setDirection(ccui.ScrollView.DIR_HORIZONTAL);
10 }
```

第9行代码设置了滚动视图的滚动方向，其相关API如表10-25所示。

表10-25　ccui.ScrollView中方向相关API

参数或返回类型	函　　数	说　　明
Number	getDirection()	获取方向
Number	setDirection(dir)	设置方向

其中，方向可取值如下。

- ❑ **ccui.ScrollView.DIR_NONE**：无方向。
- ❑ **ccui.ScrollView.DIR_VERTICAL**：竖直方向。
- ❑ **ccui.ScrollView.DIR_HORIZONTAL**：水平方向。
- ❑ **ccui.ScrollView.DIR_BOTH**：水平和竖直方向。

滚动视图在滚动的时候，可以添加一些效果，例如惯性、弹力回弹，API如表10-26所示。

表10-26　ccui.ScrollView中惯性、弹力效果相关API

返回值类型	参数类型	函　数	说　明
Boolean	无	isInertiaScrollEnabled()	获取是否开启滚动惯性效果的结果
无	Boolean	setInertiaScrollEnabled(enabled)	设置是否开启滚动惯性效果
Boolean	无	isBounceEnabled()	获取是否开启弹力效果的结果
无	Boolean	setBounceEnabled(enabled)	设置是否开启弹力效果

接下来，我给之前的滚动视图添加一张图片，代码如下：

```
1  // 加载[滚动视图]
2  loadScrollView : function(){
3      var scrollView = new ccui.ScrollView();
4      // ......
5      var imageView = new ccui.ImageView("res/unit07_design/background.png");
6      scrollView.addChild(imageView);
7      scrollView.setInnerContainerSize(cc.size(imageView.width, imageView.height));
8      imageView.x = imageView.width / 2;
9      imageView.y = imageView.height / 2;
10 }
```

第7行代码设置了滚动视图内部容器的大小，这和setContentSize(size)是有区别的。setContentSize设置的是滚动视图的大小，而setInnerContainerSize设置的是内部容器的大小。另外需要注意的是，内部容器的大小必须大于或等于滚动视图的大小。与容器以及大小相关API如表10-27所示。

表10-27　ccui.ScrollView中容器以及大小相关API

返回值类型	参数类型	函　数	说　明
ccui.Layout	无	getInnerContainer()	获取滚动视图的内部布局容器
cc.Size	无	getInnerContainerSize()	获取内部容器的大小
无	cc.Size	setInnerContainerSize(size)	设置内部容器的大小

在实际开发中，很少去监听滚动视图的滚动事件，但是这些事件接口都是存在的，如表10-28所示。值得注意的是，onTouchBegan、onTouchMoved、onTouchEnded以及onTouchCancelled函数为ccui.ScrollView内部实现用，不建议重写它们。

表10-28　ccui.ScrollView中事件相关接口

返回值类型	参数类型	函　　数	说　　明
Boolean	cc.Touch cc.Event	onTouchBegan(touch, event)	触摸开始时的回调函数
无	cc.Touch cc.Event	onTouchMoved(touch, event)	触摸移动时的回调函数
无	cc.Touch cc.Event	onTouchEnded(touch, event)	触摸结束时的回调函数
无	cc.Touch cc.Event	onTouchCancelled(touch, event)	触摸取消时的回调函数
无	Function Object	addEventListener(selector, target)	添加事件监听器

有的时候，你可能需要滚动视图滚动到某个位置，而ccui.ccui.ScrollView提供了如表10-29所示的相关API。

表10-29　ccui.ScrollView中滚动相关API

返回值类型	参数类型	函　　数	说　　明
无	Number Boolean	scrollToBottom(time, attenuated)	滚动到滚动视图的底部
无	Number Boolean	scrollToTop(time, attenuated)	滚动到滚动视图的顶部
无	Number Boolean	scrollToLeft(time, attenuated)	滚动到视图的左端
无	Number Boolean	scrollToRight(time, attenuated)	滚动到视图的右端
无	Number Boolean	scrollToTopLeft(time, attenuated)	滚动到视图的左上角
无	Number Boolean	scrollToTopRight(time, attenuated)	滚动到视图的右上角
无	Number Boolean	scrollToBottomLeft(time, attenuated)	滚动到视图的左下角
无	Number Boolean	scrollToBottomRight(time, attenuated)	滚动到视图的右下角
无	Number Number Boolean	scrollToPercentVertical (percent, time, attenuated)	按百分比竖直滚动
无	Number Number Boolean	scrollToPercentHorizontal (percent, time, attenuated)	按百分比水平滚动
无	Number Number Boolean	scrollToPercentBothDirection (percent, time, attenuated)	竖直方向和水平方向分别按一定的百分比滚动

这些接口都至少接受两个参数：time和attenuated。其中，time表示持续时间，attenuated表示滚动速度是否衰减，也就是说，这些滚动会持续一段指定的时间。当然，ccui.ScrollView还提供了瞬时就能完成的“滚动”，这里将这一动作称为跳跃，相关API如表10-30所示。

表10-30 ccui.ScrollView中跳跃相关API

返回值类型	参数类型	函　数	说　明
无	无	jumpToBottom()	跳到视图底端
无	无	jumpToTop()	跳到视图顶端
无	无	jumpToLeft()	跳到视图左边
无	无	jumpToRight()	跳到视图右边
无	无	jumpToTopLeft()	跳到视图左上角
无	无	jumpToTopRight()	跳到视图右上角
无	无	jumpToBottomLeft()	跳到视图左下角
无	无	jumpToBottomRight()	跳到视图右下角
无	Number	jumpToPercentVertical(percent)	竖直方向跳到指定百分比位置
无	Number	jumpToPercentHorizontal(percent)	水平方向跳到指定百分比位置
无	Number	jumpToPercentBothDirection(percent)	竖直和水平方向跳到指定百分比位置

10.11 ccui.PageView 分页视图

分页视图（ccui.PageView）也是直接继承自ccui.Layout，与滚动视图不一样的是，分页视图将其内部的UI控件进行分页布局，每次只能显示一页，但是可以在多页之间进行切换。

创建一个ccui.PageView的代码如下：

```
1  // 加载[PageView]
2  loadPageView : function(){
3      var pageView = new ccui.PageView();
4      this.addChild(pageView);
5      pageView.setTouchEnabled(true);
6      pageView.setContentSize(cc.winSize);
7      pageView.x = (cc.winSize.width - pageView.width) / 2;
8      pageView.y = (cc.winSize.height - pageView.height) / 2;
9  }
```

创建完一个分页后，可以通过如表10-31所示的API给分页视图添加、删除和获取页面。

表10-31 ccui.PageView中添加、删除、查询页的相关API

返回值类型	参数类型	函　数	说　明
无	ccui.Layout	addPage(page)	往分页视图的最后插入一页
无	ccui.Widget Number Boolean	addWidgetToPage(widget, pageIdx, forceCreate)	把一个UI控件添加到分页视图中
无	ccui.LayoutNumber	insertPage(page, index)	在指定位置插入一页
无	ccui.Layout	removePage(page)	移除一页
无	Number	removePageAtIndex(index)	移除指定位置的页
无	无	removeAllPages()	移除所有页
Number	无	getCurPageIndex()	获取当前显示页的索引号
Array	无	getPages()	获取所有页的列表
ccui.Layout	Number	getPage(index)	获取指定的页

（续）

返回值类型	参数类型	函　　数	说　　明
无	Number	scrollToPage(index)	滚动到一个指定的页

分页视图默认需要滑动到页面宽度大小一半的时候才会切换到下一页，但实际上，你可以设置这个滑动临界值，相关API如表10-32所示。

表10-32　**ccui.PageView**中滑动临界值相关的API

返回值类型	参数类型	函　　数	说　　明
无	Number	setCustomScrollThreshold(threshold)	设置自定义滚动临界值
Number	无	getCustomScrollThreshold()	获取自定义滚动临界值
无	Boolean	setUsingCustomScrollThreshold(flag)	设置是否使用自定义滚动临界值
Boolean	无	isUsingCustomScrollThreshold()	获取是否使用自定义滚动临界值的结果

现在我创建3张图片，把它们添加到之前创建的分页视图pageView中，代码如下：

```
1  // 加载[PageView]
2  loadPageView : function(){
3      var pageView = new ccui.PageView();
4      // ......
5      for (var i = 0; i < 3; ++i) {
6          var layout = new ccui.Layout();
7          pageView.addPage(layout);
8
9          var imageView = new ccui.ImageView("res/unit07_design/background.png");
10         layout.addChild(imageView);
11         imageView.x = pageView.width / 2;
12         imageView.y = pageView.height / 2;
13     }
14     pageView.addEventListener(function(sender, type){
15         switch (type) {
16             case ccui.PageView.EVENT_TURNING:
17                 var index = pageView.getCurPageIndex() + 1;
18                 cc.log("当前第" + index + " 页");
19                 break;
20             default:
21                 break;
22         }
23     }, this);
24 }
```

第6行代码创建了一个ccui.Layout布局对象。注意，这里必须要先给ccui.PageView添加一个ccui.Layout布局对象，这是因为在Native上，ccui.PageView只接受ccui.Layout作为参数。第7行把ccui.Layout对象添加到pageView分页视图中，第10行代码把创建的ccui.ImageView对象添加到ccui.Layout里。

第14行到第23行代码，给pageView添加了事件监听器。当分页视图有滑动事件时，则会触发ccui.PageView.EVENT_TURNING事件。

10.12 ccui.ListView 列表视图

列表视图（ccui.ListView）继承自ccui.ScrollView，所以它同样具有滚动特性。列表视图容器中的UI控件以项（item）为单位，项可以基于某个模型，然后通过克隆的方式高效利用。显然，像游戏包裹这样的功能用列表视图实现是不二的选择。

另外需要说明的是，ccui.ListView在创建元素的时候，如果元素个数非常多，需要考虑在移动ListView的过程中，动态调整项的位置来起到优化作用。比如，一个列表视图需要有100个项，但它一共可以显示10个项，此时可以多创建项，例如12个或者15个，然后在移动过程中去修改第12个项或者第15个项的位置以及其他属性。但倘若你直接创建了100个项，那么将会非常消耗内存并且影响ccui.ListView的效率。

下面的代码创建了一个列表视图：

```
1  // 加载[ListView]
2  loadListView : function(){
3      var listView = new ccui.ListView();
4      this.addChild(listView);
5      listView.setTouchEnabled(true);
6      listView.setContentSize(cc.size(240, 120));
7      listView.setDirection(ccui.ScrollView.DIR_HORIZONTAL);
8      listView.setBackGroundColor(cc.color(128, 128, 128));
9      listView.setBackGroundColorType(ccui.Layout.BG_COLOR_SOLID);
10     listView.x = (cc.winSize.width - listView.width) / 2;
11     listView.y = (cc.winSize.height - listView.height) / 2;
12 }
```

ccui.ListView中项操作的相关API如表10-33所示。

表10-33 ccui.ListView中项操作的相关API

返回值类型	参数类型	函　数	说　明
无	ccui.Widget	setItemModel(model)	设置项的模型
无	无	pushBackDefaultItem()	插入一个默认项(通过克隆模式创建)到列表视图的尾部
无	Number	insertDefaultItem(index)	插入一个默认项(通过克隆模式创建)到列表视图的指定位置
无	ccui.Widget	pushBackCustomItem(item)	插入一个自定义项到列表视图的尾部
无	ccui.Widget Number	insertCustomItem(item, index)	插入自定义UI控件到列表视图中指定索引处
无	Number	removeItem(index)	删除给定索引的项
无	无	removeLastItem()	删除最后一个项
无	无	removeAllItems()	删除所有项
Number	无	getItem(index)	获取给定索引的项
Array	无	getItems()	获取所有项
Number	无	getIndex(item)	获取指定项的索引
Number	无	getCurSelectedIndex()	获取当前所选项的索引

设置项的模型后，将该模型作为一个蓝图，然后调用pushBackDefaultItem函数，将克隆出的新副本插入到列表视图中。

在ccui.ListView中，还可以设置每个项之间的边距、对齐方式以及方向，相关API如表10-34所示。

表10-34　ccui.ListView中边距、对齐方式以及方向相关的API

返回值类型	参数类型	函　　数	说　　明
Number	无	getItemsMargin()	获取每个项之间的边距
无	Number	setItemsMargin(margin)	设置每个项之间的边距
Number	无	getGravity(gravity)	获取对齐方式
无	Number	setGravity(gravity)	设置对齐方式
Number	无	getDirection()	获取方向
无	Number	setDirection(dir)	设置方向

其中，方向的可取值和ccui.ScrollView一致，见10.9节。此外，ccui.ListView还有一些刷新相关的API，如表10-35所示。

表10-35　ccui.ListView中刷新相关的API

返回值类型	参数类型	函　　数	说　　明
无	无	requestRefreshView()	请求刷新视图和布局
无	无	refreshView()	刷新列表视图的视图
无	无	forceDoLayout()	强制刷新控件的布局
无	无	doLayout()	请求刷新控件的布局

下面给之前创建的listView添加几个项目：

```
1  // 加载[ListView]
2  loadListView : function(){
3      var listView = new ccui.ListView();
4      // ......
5      // 创建按钮
6      var default_button = new ccui.Button();
7      default_button.setName("TextButton");
8      default_button.setTouchEnabled(true);
9      default_button.loadTextures("res/unit10_ui/btn_normal.png", "res/unit10_ui/
       btn_ressed.png", "");
10     // 创建默认模型
11     var default_item = new ccui.Layout();
12     default_item.addChild(default_button);
13     default_item.setTouchEnabled(true);
14     default_item.setContentSize(default_button.getContentSize());
15     default_button.x = default_item.width / 2;
16     default_button.y = default_item.height / 2;
17     // 设置模型
```

```
18      listView.setItemModel(default_item);
19      // 添加4个项
20      for (var i = 0; i < 4; ++i) {
21          listView.pushBackDefaultItem();
22      }
23      // 给前4个项设置文本标签
24      for (var i = 0; i < 4; i++){
25          var item = listView.getItem(i);
26          var button = item.getChildByName("TextButton");
27          var index = listView.getIndex(item);
28          button.setTitleText("按钮" + i);
29      }
30  }
```

第5行到第9行用于创建并配置一个按钮，作为默认模型的基础。第10行到第16行用于创建并配置默认模型。第18行设定`listView`的默认模型为`default_item`，注意它是一个`ccui.Layout`。

第19行到第22行中，我添加了4个默认项。第23行到第29行用于获取之前添加的4个项，对它们进行进一步编程，主要就是设置它们的文本标题。

最后，还可以给`listView`添加一个事件监听器，这样当点击按钮项的时候，便可以知道当前点击的是哪个按钮，代码如下：

```
1  // 加载[ListView]
2  loadListView : function(){
3      // 创建listView
4      var listView = new ccui.ListView();
5      this.addChild(listView);
6      listView.setTouchEnabled(true);
7      listView.setContentSize(cc.size(240, 120));
8      listView.setDirection(ccui.ScrollView.DIR_HORIZONTAL);
9      listView.setBackGroundColor(cc.color(128, 128, 128));
10     listView.setBackGroundColorType(ccui.Layout.BG_COLOR_SOLID);
11     listView.x = (cc.winSize.width - listView.width) / 2;
12     listView.y = (cc.winSize.height - listView.height) / 2;
13     listView.addEventListener(function (sender, type) {
14         switch (type) {
15             case ccui.ListView.EVENT_SELECTED_ITEM:
16                 cc.log("当前项索引：" + sender.getCurSelectedIndex());
17                 break;
18             default:
19                 break;
20         }
21     }, this);
22 }
```

运行上面的代码，得到的效果如图10-10所示。

10

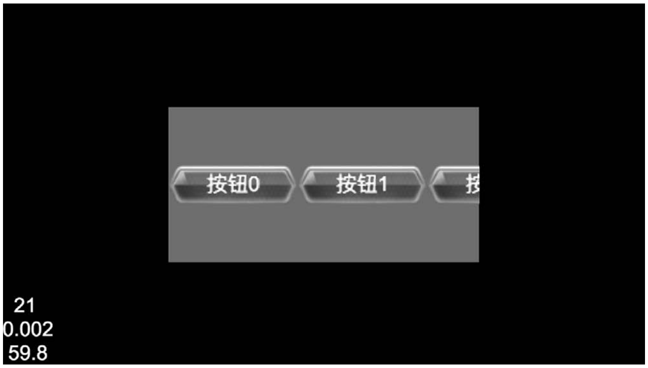


图10-10　ccui.ListView效果

10.13　实例——《保卫萝卜 2》关卡选择页面的开发

在前面的章节实例中，我们已经完成了《保卫萝卜2》的主页面开发，本章中，我们将完成《保卫萝卜2》的关卡选择页面。同样，在开发之前，先来看下本实例完成后的效果图，如图10-11所示。

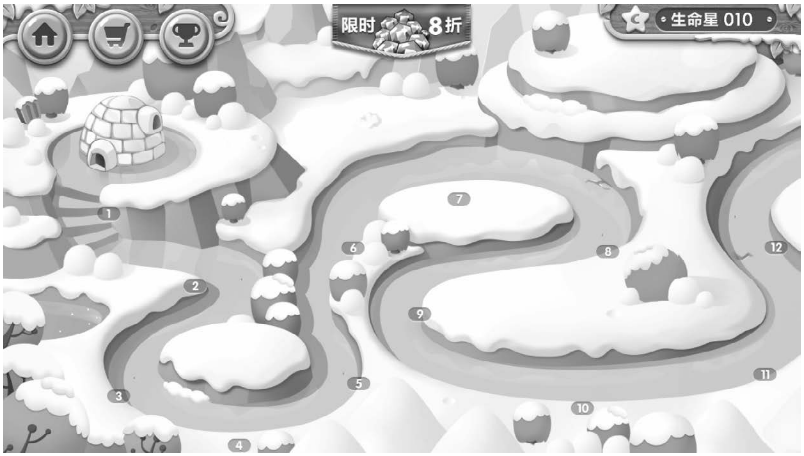


图10-11　《保卫萝卜2》关卡选择页面

分析《保卫萝卜2》的关卡选择页面，可以明显得到，我们应该把这个场景分为两个层，一个背景层，一个UI层。其中，背景层里放一个ScrollView，用来滚动整个地图。UI层存放一些简单的按钮和图片等节点。

与MainMenu场景类似，我在Carrot/src/Scene路径下新建了一个ChooseLevel文件夹，用来存放关卡选择场景中的脚本文件，文件组织结构如图10-12所示。

其中，ChooseLevel.js为场景组织文件，我在这个文件中定义了ChooseLevelScene场景，ChooseLevelScene又加载了两个层，分别是背景层和UI层，代码如下：

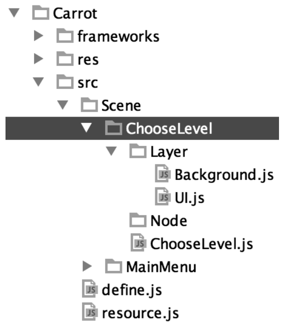


图10-12　关卡选择场景中的文件组织结构

```
1  var ChooseLevelScene = cc.Scene.extend({
2      backgroundLayer : null,
3      uiLayer         : null,
4      onEnter : function () {
5          this._super();
6          // 加载[背景层]
7          this.loadBackgroundLayer();
8          // 加载[主层]
9          this.loadMainLayer();
10     },
11     loadBackgroundLayer : function(){
12         this.backgroundLayer = new CLBackgroundLayer();
13         this.addChild(this.backgroundLayer);
14     },
15     loadMainLayer : function(){
16         this.uiLayer = new CLUILayer();
17         this.addChild(this.uiLayer);
18     }
19 });
```

第12行和第16行分别创建了背景层和UI层，这两个层的代码实现在src/Scene/ChooseLevel/Layer中的Background.js和UI.js文件里。

Background.js 文件中定义了 `CLBackgroundLayer` 类，`CLBackgroundLayer` 继承自 `cc.Layer`。根据前面的分析得出，在`CLBackgroundLayer`中需要一个`ScrollView`对象来滚动整个背景地图，接下来就来创建此`ScorllView`对象，代码如下：


```
1  // 加载[滚动视图]
2  loadScrollView : function(){
3      var node = new ccui.ScrollView();
4      this.addChild(node);
5      this.scorllView = node;
6      node.setDirection(ccui.ScrollView.DIR_HORIZONTAL);
7      node.setTouchEnabled(true);
8      node.setContentSize(cc.winSize);
9
10     var nextPosX = 0;
11     var imageView = null;
12     for (var i = 0; i < 14; i++) {
13         imageView = new ccui.ImageView("res/ChooseLevel/Map/stage_map_" + i +
           ".png");
14         node.addChild(imageView);
15         imageView.setAnchorPoint(cc.p(0, 0.5));
16         imageView.setPosition(nextPosX, cc.winSize.height / 2);
17         nextPosX += imageView.width;
18     }
19     node.setInnerContainerSize(cc.size(nextPosX, cc.winSize.height));
20 }
```

第3行创建了ScrollView对象，第6行设定了此ScrollView的滚动方向为水平方向，第7行开启了ScrollView对象的触摸开关，第8行设置了ScrollView的内容大小为整个窗口的大小。

第10行到第18行创建出地图背景图片。在ScrollView中，子节点的坐标需要我们自己设定，这里通过一个for循环干净利索地添加了14张背景图。第19行设定了ScrollView的布局容器大小，这里nextPosX的值等于14张背景图宽度的总和。

最后，在CLBackgroundLayer类的onEnter函数中，调用loadScrollView函数即可。接下来，看看UI层的开发。

显然，UI层的实现代码在UI.js文件中。UI层中需要开发的是左上角的三个按钮、中间的打折信息按钮以及右上角的生命星显示，如图10-13所示。



图10-13　关卡选择场景的顶部按钮

可以明显看出，左上角三个按钮的后面有一张背景图片，这张背景图始终位于当前屏幕的左上角，所以可以将它的锚点设置为cc.p(0, 1)，坐标设为cc.p(0, cc.winSize.height)，代码如下：

```
1  // 加载[左上角按钮]
2  loadTopLeftButtons: function(){
3      var leftPanel = new ccui.ImageView("res/ChooseLevel/stagemap_toolbar_leftbg.
       png");
4      this.addChild(leftPanel);
5      leftPanel.setAnchorPoint(0, 1);
6      leftPanel.setPosition(0, cc.winSize.height);
7      // ......
8  }
```

上面的代码保证了3个按钮的背景始终在屏幕的左上角，那么剩下的这3个按钮应该作为谁的子节点呢？当前层吗？不是，而是应该作为背景图片的子节点，使得三个按钮和按钮背景图片“绑”成了一个整体，这样的相对布局保证了3个按钮的坐标不会随屏幕大小的不一致而产生位置错乱，代码如下：

```
1  // 加载[左上角按钮]
2  loadTopLeftButtons: function(){
3      var leftPanel = new ccui.ImageView("res/ChooseLevel/stagemap_toolbar_leftbg.
       png");
4      // ......
5
6      // 加载[首页按钮]
7      this.loadHomeButton(leftPanel);
8      // 加载[商店按钮]
9      this.loadShopButton(leftPanel);
10     // 加载[排行榜按钮]
11     this.loadRankingButton(leftPanel);
```

```
12 },
13 // 加载[首页按钮]
14 loadHomeButton : function(parent){
15     var node = new ccui.Button();
16     parent.addChild(node);
17     var textures = "res/ChooseLevel/stagemap_toolbar_home.png";
18     node.loadTextures(textures, textures, "");
19     node.setPressedActionEnabled(true);
20     node.setZoomScale(0.2);
21     node.setAnchorPoint(0, 0);
22     node.setPosition(10, 10);
23     node.addTouchEventListener(function (sender, type) {
24         if (type == ccui.Widget.TOUCH_ENDED) {
25             cc.director.runScene(new MainMenuScene());
26         }
27     }, this);
28 }
```

通过第17行和第18行的代码可以看出，“首页”按钮仅采用一张图片，所以我们应该给按钮启用被按下时的缩放操作（第19行），并设置按下的缩放比例（第20行）。最后，第23行到第27行给“首页”按钮添加了触摸事件监听器，实现了主页面场景`MainMenuScene`的切换。另外，第24行代码用于判断按钮事件是否在触摸结束之后才响应实现逻辑代码，在开发中更多的按钮事件响应都应该放在触摸结束之后，而不是触摸释放之前。

“商店”和“排行榜”按钮的开发和“首页”按钮几乎一致，这里我不再贴出代码，你可以自行参考随书代码。

在中间促销按钮中，需要添加一个`TextBMFont`来表示当前打多少折扣。你可以给当前层定义一个变量，例如`discountText`，然后让它指向这个`TextBMFont`，这样在其他函数中便可以通过`discountText`控制显示的折扣，代码如下：

```
1  //加载[中间促销按钮]
2  loadDiscountButton: function () {
3      var button = new ccui.Button();
4      this.addChild(button);
5      var resourceStr = "res/ChooseLevel/zh/discount_tag_stone.png";
6      button.loadTextures(resourceStr, resourceStr, "");
7      button.setAnchorPoint(0.5, 1);
8      button.setPosition(cc.winSize.width / 2, cc.winSize.height);
9      button.addTouchEventListener(function (sender, type) {
10         if (type == ccui.Widget.TOUCH_ENDED) {
11             cc.log("点击促销按钮");
12         }
13     }, this);
14     // 折扣显示
15     var text = new ccui.TextBMFont("8", "res/ChooseLevel/discount.fnt");
16     this.discountText = text;
17     button.addChild(text);
18     text.setAnchorPoint(0, 0);
19     text.setPosition(145, 60);
20 }
```

右上角的生命星组成与中间限时折扣按钮类似，最底层为一张背景图，顶层为生命星数量文本，实现起来也较为简单，具体可见随书代码。

最后，在CLUILayer的构造函数中调用这些加载方法，再运行ChooseLevelScene场景，可以得到如图10-11所示的《保卫萝卜2》关卡选择页面。

至此，我们已经开发了主页面场景MainMenuScene和关卡选择场景ChooseLevelScene。虽然ChooseLevelScene还未开发完毕,但是已经可以将两个场景串联起来,从而实现场景的切换。你应该还记得，在《保卫萝卜2》的主页面场景中有个“开始冒险”菜单按钮，点击此菜单按钮，可以切换到关卡选择场景。此菜单按钮的实现代码位于MMMainLayer的loadMenu函数中，找到代码之后，在“开始冒险”菜单按钮的事件响应函数中添加场景切换代码（详见下面的第11行）：

```
1  // 加载[开始冒险]和[天天向上]菜单
2  loadMenu : function(){
3      // ......
4      var start = new cc.MenuItemSprite(
5          startNormal,
6          startPress,
7          startDisabled,
8          function(){
9              cc.audioEngine.playEffect(res.sd_mm_Select_mp3);
10             cc.log("点击“开始冒险”按钮");
11             cc.director.runScene(new ChooseLevelScene());
12         }.bind(this));
13 }
```

10.14 小结

通过本章的学习，我们知道了在Cocos2d-JS中有个UI模块，其命名空间为ccui，它提供的UI控件有文本、按钮、复选框、滑块、图片视图、加载条、编辑框、布局、滚动视图、分页视图以及列表视图，学习了这些UI控件的法用，了解了它们的API。在最后的章节实例中，初步开发了《保卫萝卜2》的关卡选择场景。

10.15 参考资源

本章的参考资源为Cocos引擎API：http://api.cocos.com/。

第 11 章

性能优化

性能优化是游戏开发中必不可少的一门技能，性能优化技巧直接考量一个程序员的能力所在。一个好的程序员不仅能写一手好代码，更应该熟悉各种各样的优化技巧。本章将围绕Cocos2d-JS游戏的性能优化方案展开，讲解Cocos2d-JS游戏性能优化技巧。

本章内容：

- 对象缓冲池（`cc.pool`）
- 批量渲染（`SpriteBatchNode`）
- 烘培层（Bake Layer）
- SpriteSheet资源优化

11.1 对象缓冲池

在游戏中，有些对象常常被不断创建和销毁，例如*MoonWarriors*中的子弹、敌机等，这类对象高频地创建和销毁会给游戏带来极大的性能消耗，这显然存在一定的优化空间。好比说，我们每次吃饭需要花费15分钟，家里直接取筷子只需20秒，但倘若你每次吃饭要去超市买新筷子，那我估计等你买完筷子回来的时候，你妈妈早已把你的饭菜给倒了，因为这是一个极其愚蠢且不理智的做法。所以，你应该试想，在Cocos2d-JS中是否存在一种技术方案，可以将子弹、飞机等这些经常需要被创建和销毁的对象在创建之后，就将它们缓存起来，当需要用的时候，直接从缓存中取出，不需要的时候放回缓存待命即可，这样就可以优化高频创建和销毁所带来的性能消耗。

然而答案是存在的，Cocos2d-JS提供了一个对象缓冲池（`cc.pool`对象）供我们解决此类问题。

说明 `cc.pool`对象缓冲池适合管理那些高频创建和销毁的对象，例如子弹、敌人对象，而游戏背景等这类非“活跃”对象则不适用。若要使用`cc.pool`，则需要在project.json的`modules`数组中添加ccpool项。

11.1.1　让我们的类支持 cc.pool 管理

对象的缓存以及取出都由cc.pool来管理，要想让类支持cc.pool管理，就必须实现reuse和unuse这两个方法，这是因为，对象缓存通过cc.pool.putInPool(object)函数实现，该函数会去调用该对象的unuse方法。一般情况下，我们会在unuse方法中完成对象被放入缓冲池前的操作。

reuse则是当我们要从对象缓冲池中取出对象时所触发的函数，我们可以在这里完成对象的重新初始化，让它变为可用状态。简单来说，可以将unuse理解为“死亡”而非销毁，将reuse理解为“复活”而非创建。下面我们写一个类，使其支持cc.pool管理：

```
1  var SupportPoolHero = cc.Sprite.extend({
2      _hp : 0,     // 血量
3      _sp : 0,     // 愤怒
4      _mp : 0,     // 魔法
5      ctor: function (hp, mp, sp) {
6          this._super(res.sh_node_64_png);
7          this.initData(hp, mp, sp);
8      },
9      initData: function (hp, mp, sp) {
10         this._hp, this._mp, this._sp = hp, mp, sp;
11     },
12     unuse: function () {
13         this._hp = this._mp = this._sp = 0;
14         this.retain(); // 使其引用计数+1，防止被内存回收
15         this.removeFromParent();
16     },
17     reuse: function (hp, mp, sp) {
18         this.initData(hp, mp, sp);
19     }
20 });
21
22 SupportPoolHero.create = function (hp, mp, sp) {
23     return new SupportPoolHero(hp, mp, sp)
24 };
25
26 SupportPoolHero.reCreate = function (hp, mp, sp) {
27     // 判断[对象缓冲池中是否有可用对象]
28     if (cc.pool.hasObject(SupportPoolHero)){
29         // 从对象缓冲池中获取对象
30         return cc.pool.getFromPool(SupportPoolHero, hp, mp, sp);
31     }else{
32         // 创建一个新的对象
33         return  SupportPoolHero.create(hp, mp, sp);
34     }
35 };
```

在上述代码中，我们已经定义好了一个支持cc.pool的类。在上述代码的第28行中，我们看到了一个cc.pool的函数。实际上，cc.pool暴露出给用户调用的函数并不多，只有5个，如表11-1所示。

表11-1　cc.pool暴露出给用户调用的函数

函 数 名	说 明
cc.pool.putInPool(obj);	将对象放入对象缓冲池中
cc.pool.hasObject(objClass);	判断回收池中是否有可用的指定对象
cc.pool.removeObject(obj);	删除回收池中的指定对象
cc.pool.getFromPool(objClass);	从回收池回收指定对象
cc.pool.drainAllPools();	清空对象缓冲池

一般情况下，cc.pool作用于游戏玩法场景，所以，当游戏切换到UI场景时，cc.pool中缓存的对象不再需要，此时为了避免不必要的内存占用，应该调用cc.pool.drainAllPools()清空对象缓冲池。

11.1.2　使用 cc.pool 管理对象

使用cc.pool管理对象，具体可分为以下三步。

第一步：条件配置——让我们的类支持cc.pool，这已经在11.1.1节中实现了。

第二步：预备工作——首次创建好一定量的对象，并把它们放到对象缓冲池中。创建一定量的对象非常合情合理，因为我们必须保证缓冲池和外界需要的供求关系，若缓冲池中对象不足，则该类会创建新的对象实例，并将其放入对象缓冲池中。预备工作的代码如下：

```
1  // 准备工作[创建好对象，并把它们放入对象缓冲池中]
2  prepareHeroForPool : function(){
3      for (var i = 0; i < 3000; i++) {
4          var node = SupportPoolHero.create(0, 0, 0);
5          this.addChild(node);
6          cc.pool.putInPool(node);
7      }
8  }
```

通过上述代码的第6行，我们已经成功地将创建的对象放入对象缓冲池中。因为cc.pool.putInPool()函数中调用了SupportPoolHero类的unuse()方法，而unuse()方法中实现了this.removeFromParent()函数，所以此时在场景中看不到这些对象，但它们已经被添加到了对象缓冲池中。

第三步：一切就绪——任性使用。

当预备工作准备结束之后，对象缓冲池已经有3000个SupportPoolHero对象，可以通过如下代码将SupportPoolHero对象添加到场景中：

```
1  // 从pool中获取英雄
2  addHeorByPool: function () {
3      // ......
4      for (var i = 0; i < 3000; i++) {
5          var node = SupportPoolHero.reCreate(4, 5, 6);
6          this.addChild(node);
```

11

```
7           node.setPosition(10 * i, cc.winSize.height * 0.8);
8       }
9       // ......
10 }
```

需要注意的是，在上述代码的第5行，我们通过SupportPoolHero.reCreate(args)函数创建对象，它会优先去缓冲池中取出对应的对象，若对象不存在，才会新建对象。图11-1为直接创建3000个精灵和使用pool创建3000个精灵的时间消耗对比图。

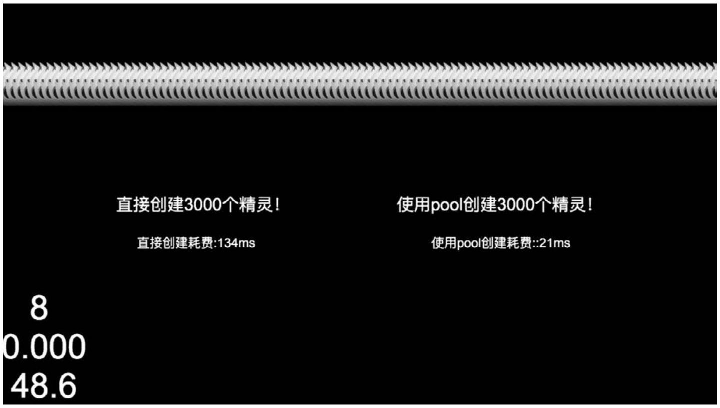


图11-1　使用CCPool获取和直接创建3000个精灵的性能消耗对比图

说明　本章的完整测试代码在随书代码res/unit11_optimize中，因部分代码较多，故不在书中一一讲解，读者可自行打开随书代码查阅或运行。

11.2　批量渲染

游戏的渲染，往往是游戏性能开销所在，其消耗了大量的资源和内存。在实际开发中，我们常常将图片按模块分类，然后用TexturePacker合成大图，这不仅可以合理地管理图片资源，也更贴近引擎的渲染优化。虽然在JSB中，底层的Cocos2d-x从3.0版本之后就提出了自动批渲染（auto-batching）的概念，但在Web上，SpriteBatchNode技术仍被推荐使用。

SpriteBatchNode为批处理绘制精灵，其原理是：创建一张纹理，然后将多个精灵添加到此纹理中，绘制时直接绘制该纹理，无需单独绘制子节点。我们都知道，在Cocos2d-x中，引擎采用OpenGL ES 2.0渲染精灵。在Web上，引擎优先采用WebGL渲染，若不支持WebGL，则采用Canvas绘制。但不管是采用OpenGL ES还是WebGL渲染精灵，其渲染每个精灵的过程都会经历“打开→绘制→关闭”的流程。也就是说，若有N个精灵，则会有N次“打开→绘制→关闭”的纹理切换流程，而通过SpriteBatchNode，绘制每个精灵的过程演变为：打开→绘制→绘制→…→关闭，游戏每一帧渲染的时候，只打开创建好的一张纹理，然后绘制N个精灵，省去了N−1次打

开和关闭的时间，从而最终实现了渲染性能优化。

11.2.1 Cocos2d-JS 中的 SpriteBatchNode 类

SpriteBatchNode 类位于 cocos2d/core/sprites/CCSpriteBatchNode.js 文件中，它继承自 cc.Node，并重写了addChild()、removeChild()、sortAllChildren()、visit()、draw()等方法。另外，addChild()方法在WebGL和Canvas渲染模式下分别被重指向相对应的函数，其指向代码（片段）如下：

```
1  if (cc._renderType === cc._RENDER_TYPE_WEBGL) {
2      // ......
3      _p.addChild = _p._addChildForWebGL;
4      // ......
5  } else {
6      // ......
7      _p.addChild = _p._addChildForCanvas;
8      // ......
9  }
```

在_addChildForWebGL()方法中，SpriteBatchNode限制了其子节点只能是精灵或者精灵的子类，且子节点和SpriteBatchNode必须使用同一个纹理（Texture2D）对象，代码如下：

```
1  _addChildForWebGL: function (child, zOrder, tag) {
2      // 检查[child是否为空]
3      cc.assert(child != null, "cc.SpriteBatchNode.addChild(): child should be
       non-null");
4      // 检查[child是否是精灵或者精灵的子类]
5      if (!(child instanceof cc.Sprite)) {
6          cc.log("cc.SpriteBatchNode.addChild(): cc.SpriteBatchNode only supports
           cc.Sprites as children");
7          return;
8      }
9      // 检查[child是否使用了同一个纹理id]
10     if (child.texture != this.textureAtlas.texture) {
11         cc.log("cc.SpriteBatchNode.addChild(): cc.Sprite is not using the same
           texture");
12         return;
13     }
14
15     zOrder = (zOrder == null) ? child.zIndex : zOrder;  // 设置[zOrder值]
16     tag = (tag == null) ? child.tag : tag;              // 设置[tag值]
17
18     cc.Node.prototype.addChild.call(this, child, zOrder, tag);
19     this.appendChild(child);
20     this.setNodeDirty();
21 }
```

由于SpriteBatchNode只是绘制一张纹理，所以我们不能为每个子节点单独设置相关的绘制参数，例如反锯齿、过滤、混合等模式，并且也只能设置一个着色器程序。

说明

- 在Cocos引擎中，ParticlesSystem、Label、TiledMap等都是采用SpriteBatchNode渲染。
- SpriteBatchNode仅在WebGL渲染模式下有效，在Canvas渲染中，则其子节点都变成正常节点，不具有性能优化效果。ParticlesSystem在Canvas渲染模式下较卡，也正是因为Canvas渲染不支持SpriteBatchNode。

11.2.2 使用 SpriteBatchNode

SpriteBatchNode批处理渲染的用法较为简单，主要分为如下3步。

第一步：将纹理加载到内存。

第二步：创建SpriteBatchNode。

第三步：创建精灵并将其添加到SpriteBatchNode。

具体代码如下：

```
1  var BatchNodeLayer = cc.Layer.extend({
2      batchNode : null,
3      ctor : function(){
4          this._super();
5          // 第一步：加载[资源]
6          this.loadResource();
7          // 第二步：加载[BatchNode]
8          this.loadBatchNode();
9          // 第三步：加载[8000个精灵]
10         this.loadSomePrince();
11     },
12     // 加载[资源]
13     loadResource : function(){
14         cc.textureCache.addImage(res.u13_prince_png);
15         cc.spriteFrameCache.addSpriteFrames(res.u13_prince_plist);
16     },
17     // 加载[SpriteBatchNode]
18     loadBatchNode : function(){
19         this.batchNode = new cc.SpriteBatchNode(res.u13_prince_png);
20         this.addChild(this.batchNode);
21     },
22     // 加载[8000个精灵]并将其添加到this.batchNode中
23     loadSomePrince : function(){
24         var node = null;
25         var pos = cc.p(0, 0);
26         for (var i = 0; i < 8000; i++) {    // 创建8000个精灵
27             node = new cc.Sprite("#prince_stand_1.png");
28             this.batchNode.addChild(node);  // 添加到batchNode对象中
29
30             pos.x = Math.random() * cc.winSize.width;
31             pos.y = Math.random() * cc.winSize.height;
```

```
32              node.setPosition(pos);
33          }
34      }
35 });
```

上述代码创建了8000个精灵。图11-2和图11-3分别为使用SpriteBatchNode和没有使用SpriteBatchNode的性能对比，图11-2的帧率为60，而图11-3的帧率只有27.7。

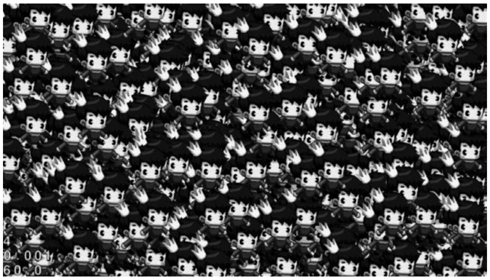


图11-2 使用SpriteBatchNode渲染

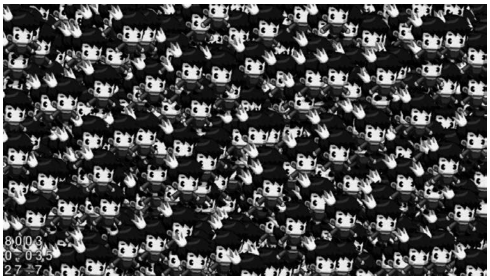


图11-3 未使用SpriteBatchNode渲染

11.3 烘培层

在一些带有关卡的游戏中，为了节约资源，通常会采用一些简约背景图片加配置装饰物的方式生成背景效果，其自身和子节点在创建完毕后，就很少发生例如添加、删除、运行动作等变动，像这类“活跃性”较低的层，Cocos2d-JS为其量身定制了一套渲染优化机制——图层预烘培。

图层预烘培优化得益于Cocos2d-JS为cc.Layer追加的bake函数。一个调用了bake函数的层会将其自身以及所有子节点烘培到一张画布（canvas）上，只要自身以及所有子节点不发生例如添加或删除子节点，以及运行动作等变动行为，那么在下一帧绘制时，引擎会将预烘培到画布上的内容直接绘制出来，从而实现渲染优化，使得原本需要调用多次绘制的层，现在只需绘制一次。

图层烘培优化技术的步骤较为简单，只需将需要烘培的节点元素添加到某一层中，然后调用该图层的bake函数即可，代码如下：

```
1  var layer = new cc.Layer();
2  var node = null;
3  var pos = cc.p(0, 0);
4  for (var i = 0; i < 8000; i++) {     // 创建8000个精灵
5      node = new cc.Sprite("#prince_stand_1.png");
6      layer.addChild(node);
7
8      // 配置[坐标属性]
9      pos.x = Math.random() * cc.winSize.width + 150;
10     pos.y = Math.random() * cc.winSize.height;
11     node.setPosition(pos);
12 }
13 layer.bake();
```

上述代码创建了8000个精灵。图11-4和图11-5分别为烘培前和烘培后的效果对比。可以明显看到，左下角的`drawCall`从8006降到了4，帧率也从25.5提升到了59.9。

另外，当图层不需要烘培时，可以通过`unbake`取消烘培。本节的详细代码见随书源码src/unit11_optimize/Layer/BakeLayer.js。

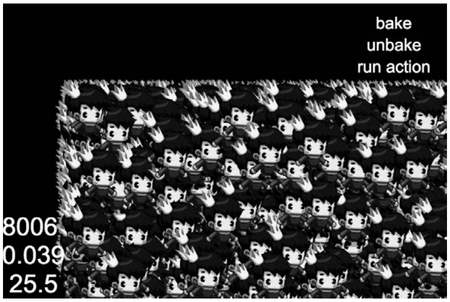


图11-4　烘培前效果

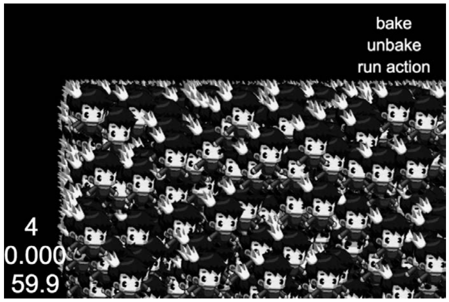


图11-5　烘培后效果

说明

- ❑ bake功能适用于“活跃性”较低的图层，对于子节点以及自身经常发生变动的层，则会给游戏带来额外的性能开销。
- ❑ bake功能仅在Canvas渲染模式下有效。在JSB以及WebGL渲染模式下调用该函数，虽不会报错，但也无任何效果。

11.4　SpriteSheet 资源优化

图片资源不仅影响游戏安装包的大小，也会占据大部分的内存空间，游戏安装包的大小直接影响用户的下载量，而内存的开销则影响游戏的性能。所以，合理地管理和优化资源，也是游戏优化的一种技术。

在实际开发中，我们通常会将图片通过TexturePacker合成大图，这不仅可以合理地管理资源，也更贴合引擎的渲染优化，例如SpriteBatchNode和Auto-Batching等。本节将介绍如何利用TexturePacker合成大图，从而缩小游戏安装包的大小以及降低游戏内存开销。

11.4.1　图片内存计算

一张图片载入内存所占用的大小为：`bytes = width * height * bitsPerPixel / 8`。其中，`bytes`表示总字节数，`width`和`height`分别表示图片的宽度和高度。`bitsPerPixel`为像素深度，表示存储每个像素所用的位数，它也是用来度量图像的分辨率的一个标准。

在OpenGL/WebGL中，纹理的宽度和高度都必须是2的n次幂。例如，一张大小为480×320的图片，在载入内存后，其纹理大小变为512×512，若大小为70×300，则纹理大小变为128×512，纹理大小始终遵循“超出，则向上取最接近的2次幂数”的原则。在Cocos2d-JS中，默认情况下，一个像素用4个字节表示，一个字节占8位，每个字节分别表示Red（红色）、Green（绿色）、Blue（蓝色）以及 Alpha（透明通道）。所以，一张像素深度为32、大小为480×320的图片所占用的内存为：

$$512 \times 512 \times 32 / 8 = 1048576\text{B} = 1024\text{KB} = 1\text{MB}$$

实际上，这样的加载方式浪费了2块大小分别为512×192和32×320的空间，如图11-6所示。

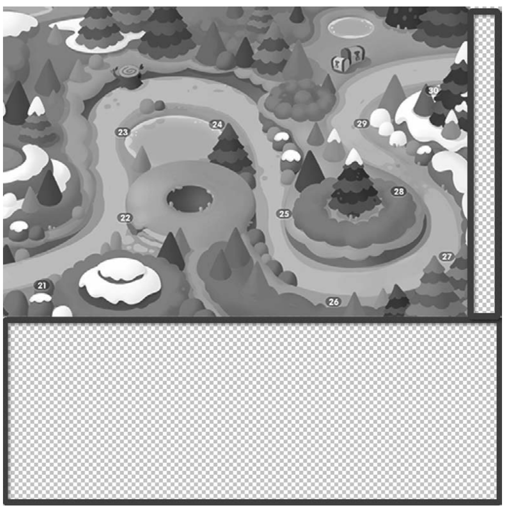


图11-6　小图直接加载的内存浪费情况



图11-7　小图合成大图

试想一下，若所有的图片，都通过这样的方式加载到内存，那么必然照成大量的内存浪费，所以在游戏开发中，我们通过TexturePacker将所有的小图合成大图（如图11-7所示），合理地规划资源，将浪费的纹理空间充分利用起来，降低了内存的开销。

11.4.2　TexturePacker

TexturePacker是一个跨平台的图片打包软件，它通过指定的图片排列算法将多张小图合成一张大图，高效充分地利用了纹理空间，大大降低了普通图片加载到内存所造成的内存浪费。TexturePacker不仅可以将小图合成大图，还可以改变大图的图片格式、对图片资源加密等。TexturePacker的下载地址为https://www.codeandweb.com/texturepacker/，图11-8为TexturePacker页面图。

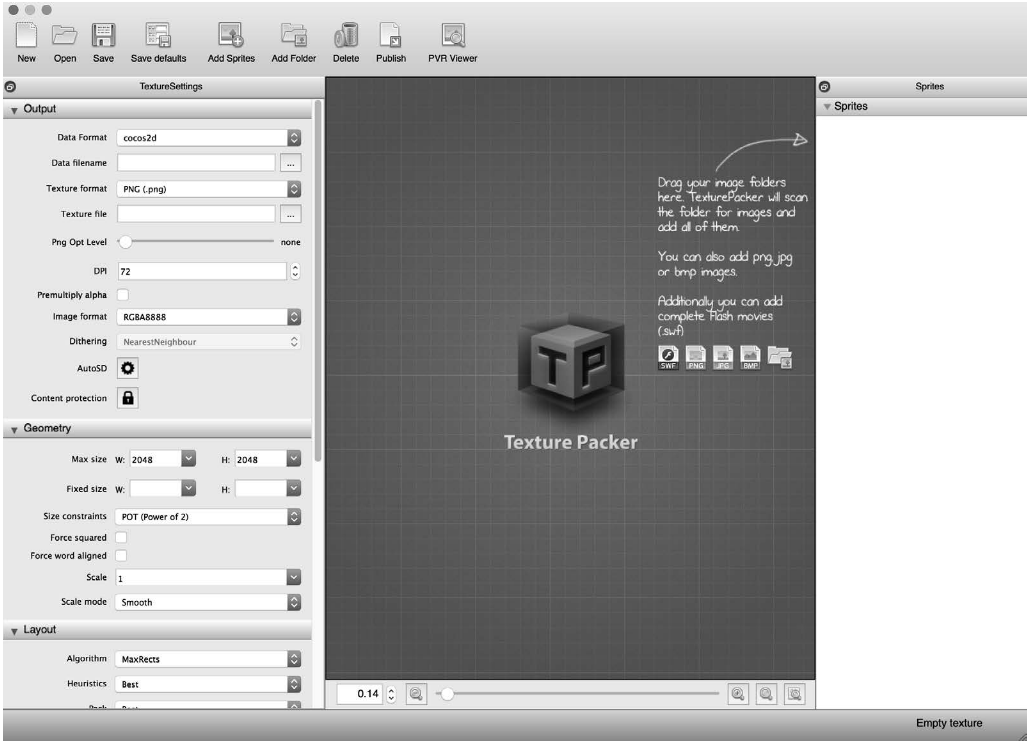


图11-8　TexturePacker页面图

1. 工具栏

和大多数软件一样，TexturePacker顶部都是工具栏，无非就是一些新建、打开、保存等基础操作。通过点击Add Sprites按钮，选取需要的图片导入。当然，也可以直接把图片或者文件夹拖曳到右侧的Sprites中。

2. TextureSettings

在图11-8中，左侧的TextureSettings是一些设置，包含输出（Output）、构图（Geometry）、布局（Layout）等模块。

在输出（Output）模块中，主要选项解释如下。

- Data Format：格式化为选中框架所支持的数据格式，当前默认选定为cocos2d。
- Data filename/Texture file：plist和纹理文件的输出路径。纹理的默认大小为2048 × 2048，这是大多数设备能够支持的最大限制。倘若我们将打包的图片合成大图后，其纹理大小超出了2048 × 2048，TexturePacker会在其右下角提示我们进行分包操作，如图11-9所示。此时勾选Layout中的Multipack复选框，然后在Data filename/Texture file输入栏中输入路径，其分包格式为xxx{n}.plist或者xxx{n}.png，如图11-10所示。



图11-9 分包操作提示

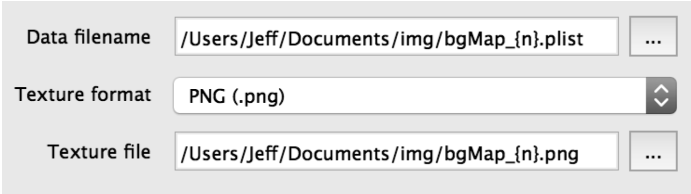


图11-10 分包之后的文件名格式

- Texture format：纹理格式化格式。常用的有PNG和PVR格式。在HTML5上，仅支持PNG格式。PVR为压缩格式，这个格式的文件是iOS设备芯片不需要经过解析就能直接读取和显示的文件。相比PNG格式，PRV拥有运行速度更快以及内存消耗更低的优势。一般情况下，PVR消耗的内存比PNG小25%左右。

  当格式选择PVR类型时，应当勾选Premultiply alpha复选框。另外，PVR格式支持图片加密，我们可以点击Output下的Content protection按钮，然后再分别点击Create new key→Save as global key→Use global key按钮，或者点击Content protection按钮后直接点击Use global key，它会一步到位，自动帮我们生成并使用密钥，如图11-11所示。注意，生成密钥后，不要忘记记录你的密钥，因为在游戏中，我们需要它来对资源进行解密。

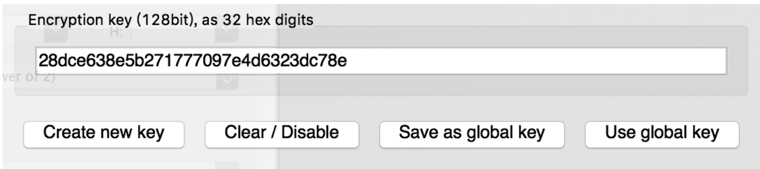


图11-11 密钥生成

在构图（Geometry）和布局（Layout）模块中，主要选项解释如下。

- Max size：设定SpriteSheet大小的最大值，默认为2048×2048。目前，大多数的设备都能支持这个大小，若设置过大，则在其他低端手机上会加载失败。所以，保持默认值即可。
- Scale：缩放系数，其取值范围为[0, 1]，其中包含0和1。当Scale系数为0.5时，TexturePacker会生成一张宽度和高度是原始图片一半的普清图片，这对于以前在不支持高清显示的设备上制作SpriteSheet非常有用。
- Algorithm：图片排列算法，默认是MaxRects，它按照图片尺寸大小排列。如果选择Basic算法，则是按行排列，当第一行排满之后，再排到第二行。
- Border/shape padding：设置在SpriteSheet里图片与图片之间的间隔。

- Extrude： 精灵边界的重复像素个数，若合成SpriteSheet后，边缘出现一两个透明像素，将此值设置大一点即可，例如1或者2。
- Trim：移除图片四周的透明区域。这使得一张SpriteSheet能更好更多地摆放图片。移除掉的透明区域在游戏中不影响，只是单纯地为了能在一张SpriteSheet摆放更多的图片。
- Shape outlines：用于打开精灵的边框，这在调试的时候极其有用。

剩余其他设定，一般保持默认值即可。

当使用TexturePacker合成SpriteSheet大图之后，会生成一个.plist配置文件和一个.png（或者pvr.ccz等）图片资源文件。在程序中，通过如下代码可将SpriteSheet大图加载到内存：

```
1  cc.textureCache.addImage(res.u13_prince_png);
2  cc.spriteFrameCache.addSpriteFrames(res.u13_prince_plist);
```

11.4.3 图片资源压缩

在HTML5上，我们必须高度重视游戏的加载速度，过长的加载速度轻则影响用户体验，重则造成用户流失。游戏的加载速度由游戏包的大小以及HTTP对图像资源的请求次数等共同决定。我们可以通过TexturePacker将大量的小图合成大图，这降低了HTTP的请求次数。而游戏包的大小，我们一般采用TinyPNG对图片进行压缩。

TinyPNG是一款来自国外的在线图片压缩云应用，其链接为https://tinypng.com/，页面如图11-12所示，我们可以将需要压缩的图片直接拖曳进去，就可自动压缩，压缩完毕后，点击download，即可下载压缩好的图片。

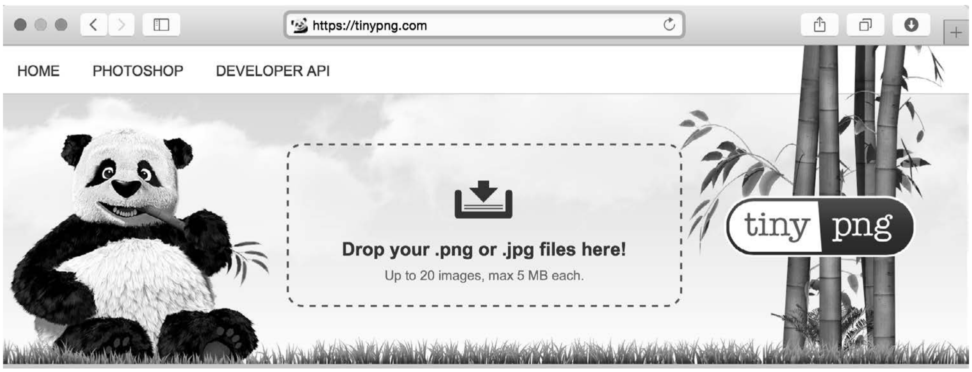


图11-12 TinyPNG页面

我试着拖曳一些图片上去压缩，压缩后如图11-13所示。我们发现，压缩率在70%左右，这是因为TinyPNG将我们上传上去的图片压缩为8位的图片，如果仔细对比压缩前和压缩后的图片，还是能发现差别的，特别是在一些渐变等过渡效果上。经过压缩的图片，质量已经下降了，所以我们需要平衡好图像大小和质量。压缩是一把利剑，虽好，但不可滥用。

stagemap_31-40.png	368.7 KB	finished	130.5 KB	download	−65%
Path53.png	116.8 KB	finished	36.2 KB	download	−69%
worldcup_10stones.png	24.8 KB	finished	8.4 KB	download	−66%
worldcup_25stones.png	25.0 KB	finished	8.5 KB	download	−66%
worldcup_ball.png	17.3 KB	finished	5.6 KB	download	−68%
worldcup_mainbg.png	508.0 KB	finished	108.0 KB	download	−79%
worldcup_shadow.png	3.8 KB	finished	494 B	download	−87%
worldcup_share.png	11.7 KB	finished	3.9 KB	download	−67%

图11-13　压缩信息

说明　关于压图技术，还有很多解决方案，TinyPNG只是其中的一种。TinyPNG有PhotoShop插件，读者可自行通过Google或百度查看。

11.5　小结

通过本章的学习，我们知道一些“高频活跃”对象可放入对象缓冲池中以便以后可以重复利用，这是在对象上的优化。而在渲染方面，OpenGL/WebGL和Canvas上都存在一套自己的优化方案。在OpenGL/WebGL中，存在一种批处理渲染的概念，主要是采用同一张纹理，优化了每次切换纹理所带来的性能开销，引擎中一些类也都采用批处理渲染，例如粒子系统、TiledMap地图等。而在Canvas上，同样存在一种称为预烘培的渲染优化方案，它主要用来将不活跃的节点都烘培到一个层上，然后一次性直接绘制。另外，在资源管理方面，我们可以通过SpriteSheet的方式，进行资源的合理管理以及内存优化等。SpriteSheet采用TexturePacker跨平台软件制作，TexturePacker也可以为我们加密图片资源。最后，我们的图片资源还可以通过TinyPNG图片压缩云应用进行压缩。其压缩比例约70%，但是会降低图片的质量，需要我们在质量和资源大小之间权衡，但不管如何，这样的压缩比例在HTML5上极具有诱惑力。

第12章 游戏地图

在一些战略类的游戏中，游戏背景地图通常会超出手机屏幕大小，并且背景地图上的很多元素都具有极高的重复性，若使用传统的方式，对于每一张背景地图，都让美术画一张图片，然后把这些图片加载到内存中，最后通过代码拼接起来，这样做的方式恐怕不太好。因为它不仅会加大美术的工作量、游戏安装包的大小、CPU的使用率、内存的占有率等，而且过高的内存开销还可能造成内存不足，引起程序崩溃。另外，这还极大降低了工作流的灵活性，使得后期背景地图需求改动时，美术、程序这一条链一起改动，这无疑加大了项目的开发成本。

所以，秉承着复用、高效、灵活的优化思想，我们可以考虑使用瓦片地图来解决和避免这一系列的问题。

本章内容：

- 瓦片地图原理
- Tiled Map Editor简介
- Tiled Map Editor工作区介绍
- 使用Tiled Map Editor制作一个地图

12.1 瓦片地图原理

瓦片地图的原理是把图片中的元素进行单位化，将大图片拆成一张张小图块，然后通过组合拼接的方式，重新拼出整个地图。其流程为：项目定好游戏中瓦片的大小，然后美术绘制瓦片并制作成瓦片图集，再由策划或美术根据项目需求通过瓦片地图编辑器制作出瓦片地图，最后将生成的地图数据文件和瓦片图集交予程序处理即可。

图12-1为美术制作好的瓦片图集，看上去似乎每个瓦片之间没有任何的关系，然而可以通过瓦片地图编辑器将其编辑为图12-2的样子，这时候，它看上去是不是就是一张有模有样的游戏地图了？



图12-1　瓦片图集

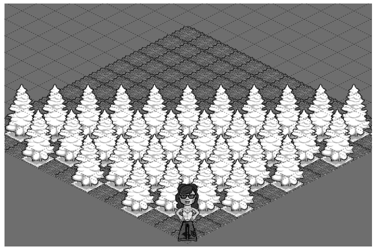


图12-2　正在编辑中的瓦片地图

12.2　Tiled Map Editor 简介

在开发中，我们采用Tiled Map Editor编辑器快速编辑游戏地图，它是一款开源的瓦片地图编辑器，采用QT编写，拥有跨平台特性，所以它能够很好地运行在Mac OS X、Windows以及Linux桌面操作系统上，其下载地址为：http://www.mapeditor.org。

Tiled Map Editor编辑器生成的地图文件是一个XML配置文件，其文件后缀名为.tmx。Cocos2d-JS友好地支持了由Tiled Map Editor制作并保存为TMX格式的地图文件。Tiled Map Editor的界面图如图12-3所示。

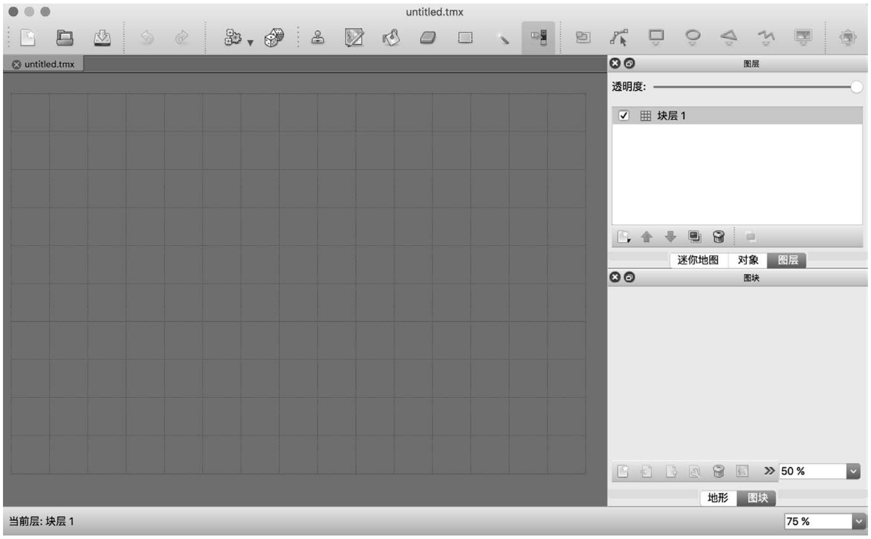


图12-3　Tiled Map Editor 界面图

说明　因为Tiled Map Editor具有跨平台特性，所以Tiled Map Editor在其他操作系统中的页面布局、使用方式等几乎一致，本章采用Mac OS X版的Tiled Map Editor讲解。

12.3　Tiled Map Editor 工作区介绍

我试着将Tiled Map Editor的工作区分为4个部分，分别是：顶部工具栏，中间地图编辑区，右上方迷你地图、对象、图层区，右下方地形和图块区。你可以先了解这些工作分区大概的作用，最后跟随本章实例进行实战。

12.3.1　工具栏

如图12-4所示，顶部工具栏除了提供新建、打开、保存等这些最基础的功能按钮外，还提供了一些快捷操作的按钮，例如图章刷、地形刷、填充、橡皮、区域选择、魔棒工具等，这些工具使用极其简单，擅用它们，会给我们带来更快、更方便的地图编辑体验。

紧随其后的则是一些形状插入等功能，例如矩形、圆形等，它们被用在对象层上，一般用来设计关卡等。



图12-4　Tiled Map Editor顶部工具栏

12.3.2　地图编辑区

顾名思义，地图编辑区即瓦片地图编辑的地方。如图12-5所示，你可以在这里制作出你想要的地图。例如，可以选中工具栏中的图章刷，然后在图块（见12.3.4节）中选一图块，即可在地图编辑区中刷出一块瓦片，接着通过按住鼠标左键后拖曳鼠标的方式，图章刷即可像刷子一样在地图编辑区域刷出大量的瓦片。另外，值得注意的是，当刷瓦片时，记得在图层（见12.3.3节）中选择你想要编辑的图层。

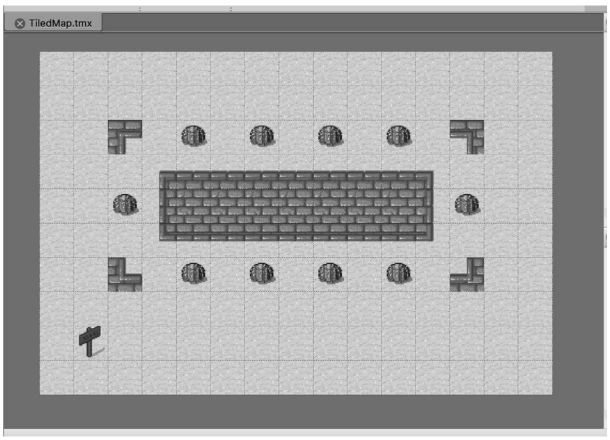


图12-5　Tiled Map Editor地图编辑区

12.3.3 迷你地图、对象、图层区

在迷你地图中，你可以像放大镜那样去观察你的瓦片地图的细节，这在一定程度上也可以快速帮你定位到想看到的区域，如图12-6所示。

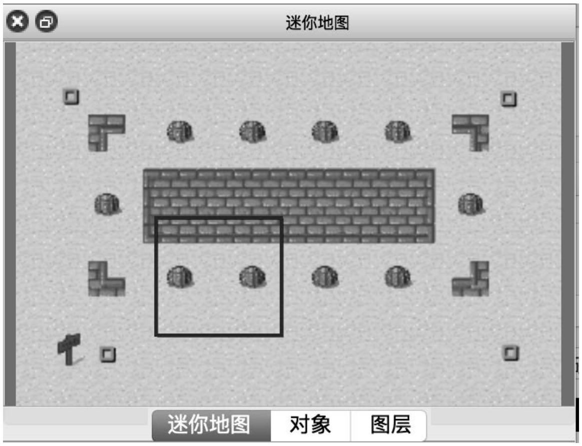


图12-6 Tiled Map Editor迷你地图

对象层实际上是一个对象数组，每个对象本身具有位置、大小等属性，如图12-7所示，它被大量应用在游戏关卡、人物移动路径、游戏道具、金币、障碍物等这类具有标记性功能的设定上。例如，塔防游戏中怪物行走的路线，跑酷游戏中路上的花样金币组等。对象组中的对象在TMX文件中以键值对的形式存在，你可以利用这个功能给对象定义一些属性，然后在代码中获取出来，如此便完成了一个简单的数据配置功能。

图层即为瓦片层，每一块瓦片都被放到这个图层中，一个地图文件中可以有多个图层，如图12-8所示。在Cocos2d-JS中，图层用`cc.TMXLayer`类表示，`cc.TMXLayer`继承自`cc.SpriteBatchNode`，每个图层使用一张`cc.TextureAtlas`来渲染瓦片，也就是美工做的瓦片集，多个图层可以共用一张瓦片集，但是一个图层不可以用多张瓦片集。



图12-7 Tiled Map Editor对象面板以及属性面板



图12-8 Tiled Map Editor图层面板

虽然TMX文件中图层的数量没有设定上限，但我们应该合理地规划好每一个图层，因为每个图层仅支持一张瓦片集（瓦片素材组）。另外，每一个瓦片用`cc.Sprite`表示，这使得我们的瓦片可以通过`cc.Sprite`类完善的API进行旋转、缩放、移动等操作。

12.3.4　图块、地形区

图块区即为瓦片的模具列表，每一个图块都有一个全局标识（GID，Global IDentifier的缩写），一般我们称它为瓦片ID。瓦片ID从1开始，起点为地图的左上角，水平向右递增，到达当前行末尾之后自动换行继续递增，如图12-9所示。另外，建议每个图块的地图素材大小为32×32的倍数。

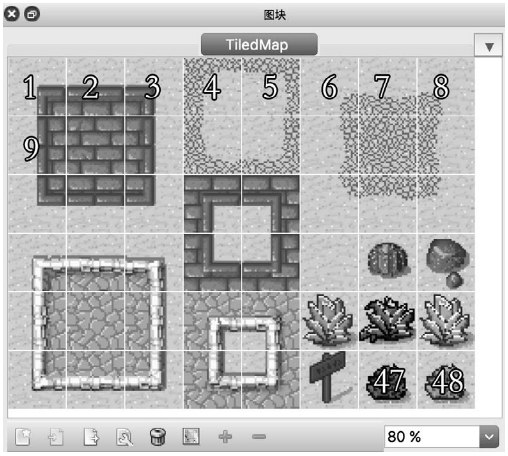


图12-9　GID

注意　在Tiled Map Editor编辑器中，GID从0开始，但是在Cocos2d-JS中，所有的GID会被执行+1操作，因为0用来表示当前位置没有瓦片。

12.3.5　Cocos2d-JS 支持的地图类型

Cocos2d-JS支持的地图类型有：直角鸟瞰地图（90°地图）、等距斜视地图（斜45°地图）和正六边形地图，不支持左右或上下边界的六边形地图。

直角鸟瞰地图和正六边形地图以垂直地图平面的方向进行展示，如图12-10和图12-11所示，它们是我们经常看到的一种地图。等距斜视地图是以倾斜地图平面45°的视角进行展示的，如图12-12所示。

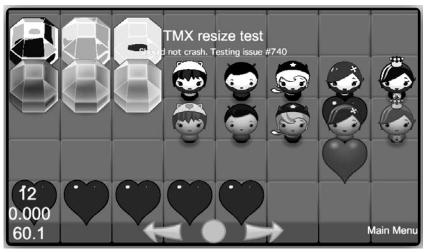


图12-10　直角鸟瞰地图



图12-11　正六边形地图

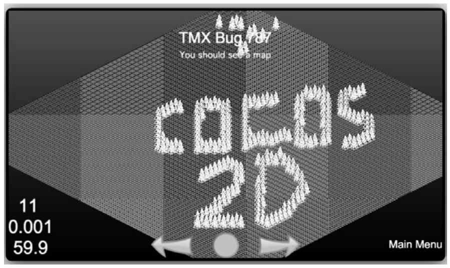


图12-12　等距斜视地图

12.3.6　瓦片地图坐标系

在瓦片地图中，左上角为瓦片地图坐标系的原点，水平向右表示X轴，垂直向下表示Y轴，每一个瓦片表示一个坐标单位。例如，在一个30行40列的瓦片地图中，坐标(1, 2)表示第2列第3行的那个瓦片。图12-13和图12-14分别为直角鸟瞰地图和等距斜视地图的坐标系统示意图。

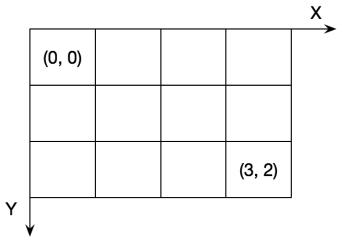


图12-13　直角鸟瞰地图的坐标系统

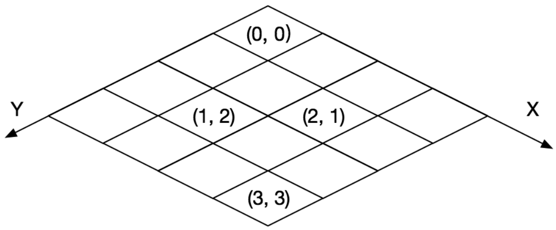


图12-14　等距斜视地图的坐标系统

12.4　使用 Tiled Map Editor 制作一个地图

在上一节中，你了解了Tiled Map Editor的工作区，那么在本节中，我将和你一起使用Tiled Map Editor制作一个瓦片地图，完成后的效果图如12-15所示。

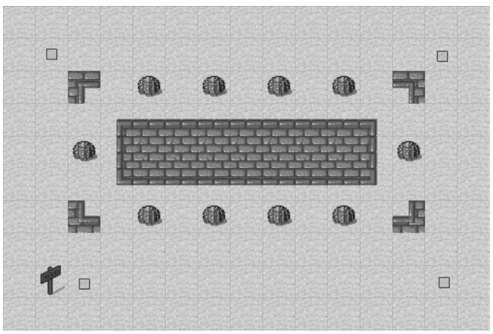


图12-15　Tiled Map Editor制作的地图

12.4.1 新建地图

打开Tiled Map Editor编辑器，点击菜单栏中的“文件”→“新建”来创建一个新地图文件，将地图大小设置为15 × 10、块大小设置为64 × 64，如图12-16所示，然后将创建的地图文件命名并保存，例如我把它命名为TileMap。



图12-16 新建地图

12.4.2 新建图块

接着，点击图块面板中的“新图块”按钮，将随书资源JSBook/res/unit12_tiledMap/TiledMap.png导入到图块中，或者也可以直接将TiledMap.png图片拖到图块面板中，再把块的宽度和高度都设定为64，边距都设置为2，如图12-17所示。

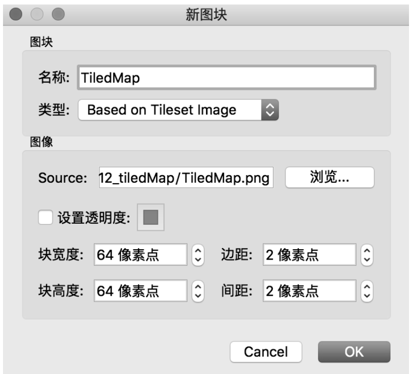


图12-17 新图块

这里将新图块中的块宽度和高度都设置为64，并不是因为之前瓦片地图的大小设定（如图12-15所示），而是因为TiledMap.png（如图12-18所示）瓦片图集中块的大小为64 × 64。

另外，美术还设定了这张图集中每个瓦片之间的边距为2个像素，她这么设计是为了隔开每个瓦片，使得图集中的瓦片排列整齐，清晰可见，不会挤到一块儿去。但是，这仅仅是美术排版用的。在程序中，我们并不希望瓦片之间有边距，这可以在新建图块的时候将这个边距去掉。

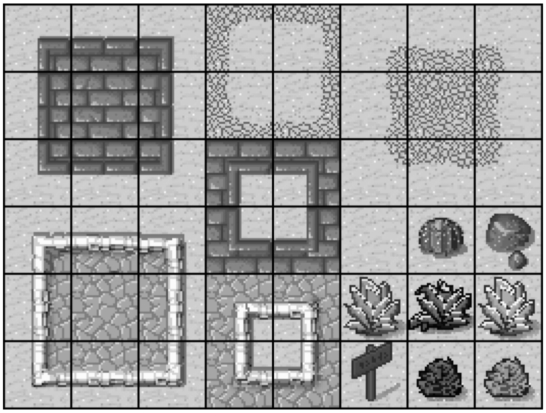


图12-18　TiledMap.png

12.4.3　新建层

从图12-15中可以看出，我们要制作的地图有3个层，分别是背景层、障碍物层以及对象层。点击图层面板中的“新图层”按钮，创建出两个普通层和一个对象，并对其命名，如图12-19所示。



图12-19　创建图层

12.4.4　编辑地图

前面提到过，Tiled Map Editor工具栏中有个填充工具，它可以用来填充层中的空白区域，这对编辑大量重复的瓦片地图非常有用。所以，我们可以使用它来编辑背景层，具体操作如下。

(1) 在TiledMap图块中选择背景用的图块。

(2) 在图层中选中bg层。

(3) 在工具栏中选择填充工具。

(4) 在地图编辑器区域中点击一下鼠标。

通过上面的步骤，就可以秒刷出背景层。接下来，通过工具栏中的图章刷工具，并选择TiledMap图块中对应的图块，然后在obstacle层中编辑出如图12-15所示的样式即可。

最后，选中hero_road对象层，然后选择工具栏的矩形，可在地图中标记出一些“标记”，这些“标记”可以用来点缀出一些图案，例如用300个“标记”画出一个爱心形状。此外，这些“标记”也可以用来标记出怪物的移动路径等，就像图12-15那样。

此时你可能会考虑一个问题：之前的图集中只有地板和植物，如果想要在这个地图中添加一只乌鸦，应该怎么做呢？做法有两种，其一，让美术修改之前的瓦片图集，然后重新编辑地图，这无疑是一种非常笨拙的方法。其二，在美术资源准备阶段，就将瓦片图集单元化，分为背景瓦片图集、障碍物瓦片图集、鸟类等瓦片图集。这样，后面再有新类型的瓦片需要添加到地图中的时候，只需将对应的瓦片图集添加到瓦块面板中，然后再新建一个新的层，例如将这个层命名为animal，接着便可以在animal层中刷出几只乌鸦，如图12-20所示。这样做的方式更为灵活，但是也不可滥用。另外，切记一个图层只能使用一张瓦片图集。

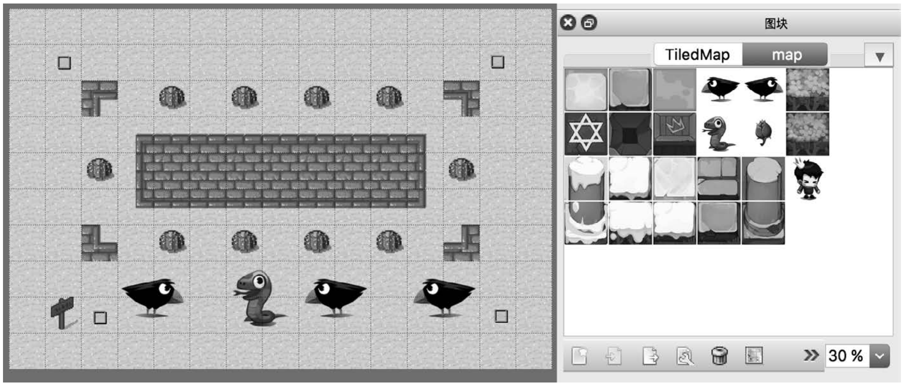


图12-20　添加动物

到此，一个瓦片地图就已经制作完成了，此时会得到一个.tmx地图信息文件以及美术事先准备好的瓦片集图片，如图12-21所示。



图12-21　编辑好的瓦片地图文件

12.5 在 Cocos2d-JS 中使用瓦片地图

在引擎中，瓦片地图相关的类主要有cc.TMXTiledMap、cc.TMXLayer和cc.TMXObjectGroup这3个类。cc.TMXTiledMap用来创建一个瓦片地图，cc.TMXLayer表示瓦片地图中的层，而cc.TMXObjectGroup则用来管理对象层数据。

12.5.1 cc.TMXTiledMap 地图对象

用Tiled Map Editor编辑器编辑完地图之后，生成一个.tmx地图文件，将此文件作为cc.TMXTiledMap构造函数的参数，即可创建出一个瓦片地图，代码如下：

```
1  loadTiledMap : function(){
2      var node = new cc.TMXTiledMap("res/unit12_tiledMap/TiledMap.tmx");
3      this.addChild(node);
4      node.setPosition((cc.winSize.width - node.width) / 2, 0);
5  }
```

实际上，cc.TMXTiledMap构造函数接收两个参数，第一个是.tmx地图信息文件，第二个是地图的瓦片图集，但是这个参数是可省略的，因为.tmx文件里面已经附带了瓦片图集的名称。

另外，cc.TMXTiledMap的锚点为cc.p(0, 0)，所以第3行代码是将瓦片地图的坐标设置为屏幕的中心。

当瓦片地图添加完毕之后，你可以通过如表12-1所示的相关API获取出层。

表12-1 cc.TMXTiledMap获取出层的相关API

返回值类型	参数类型	函　数	说　明
cc.TMXLayer	String	getLayer(layerName)	根据名字获取指定层
cc.TMXObjectGroup	String	getObjectGroup(groupName)	根据名字获取指定对象层
Array	无	allLayers()	获取所有的层
Array	无	getObjectGroups()	获取所有对象组
无	Array	setObjectGroups(groups)	设置对象组

getObjectGroup(groupName)的返回对象中存放了我们之前的路径标记，这需要遍历cc.TMXObjectGroup对象中所有的对象，把相应的数据值取出，然后来保存在数组中，代码如下：

```
1  loadTiledMap : function(){
2      var node = new cc.TMXTiledMap("res/unit12_tiledMap/TiledMap.tmx");
3      this.addChild(node);
4      node.setPosition((cc.winSize.width - node.width) / 2, 0);
5
6      var road = node.getObjectGroup("hero_road");
7      var objs = road.getObjects();
8      var roadX = null;
9      var pointArray = [];
10     for (var i = 0; i < objs.length; i++) {
11         roadX = objs[i].x + (cc.winSize.width - node.width) / 2;
```

12

```
12          pointArray.push(cc.p(roadX, objs[i].y));
13      }
14  }
```

第10行到第13行代码用于将对象组中的点坐标获取出来，并保存在pointArray数组中。你可以看到，在第11行代码中，我对每个“标记”的X轴坐标进行了调整，这是因为在屏幕适配时，我选择了FIXED_HEIGHT模式作为分辨率适配模式，它会纵向放大地图以适应屏幕的高度，横向按原始宽高比放大。所以，这里需要修正“标记”X轴的坐标，其做法为加上地图的偏移量，原理如图12-22所示。

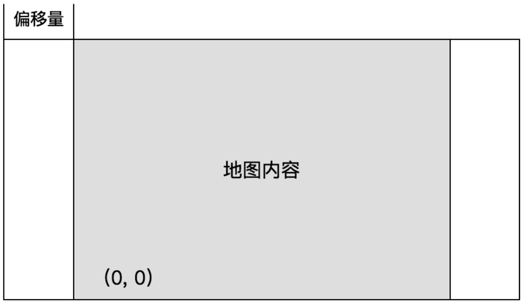


图12-22　对象数据偏移量

此外，cc.TMXTiledMap还提供了相关大小以及方向的相关API，如表12-2所示。

表12-2　**cc.TMXTiledMap**大小、方向相关的API

返回值类型	参数类型	函　　数	说　　明
cc.Size	无	getMapSize()	获取以瓦片数量为单位的地图大小
无	cc.Size	setMapSize(size)	设置以瓦片数量为单位的地图大小
cc.Size	无	getTileSize()	获取瓦片大小
无	cc.Size	setTileSize(size)	设置瓦片大小
Number	无	getMapOrientation()	获取地图方向
无	Number	setMapOrientation(orientation)	设置地图方向

需要注意的是，这里的大小和节点的尺寸不大一样。假设我们的地图是10行15列的，每块瓦片的边长是64个像素，那么调用getMapSize()后，得到的结果就是(15, 10)。可见，地图的大小是以瓦片数量为单位的。

再者，瓦片地图的瓦片是cc.Sprite类型的对象，所以瓦片的尺寸就是精灵节点的大小。前面假设瓦片变成64个像素，那么调用getTileSize()函数后，得到的结果就是cc.Size(64, 64)。

另外，瓦片地图方向的可取值如下。

- **cc.TMX_ORIENTATION_ORTHO**：直角鸟瞰地图（90°地图）。
- **cc.TMX_ORIENTATION_HEX**：正六边形地图。

- cc.TMX_ORIENTATION_ISO：等距斜视地图（斜45°地图）。

在瓦片地图中，几乎所有相关的对象都可以携带一些自定义的属性信息。关于属性，cc.TMXTiledMap提供如表12-3所示的相关API。

表12-3　cc.TMXTiledMap属性操作相关API

返回值类型	参数类型	函　数	说　明
Object	无	getProperties()	获取属性
无	Object	setProperties(properties)	设置属性
String	String	getProperty(propertyName)	通过属性名获取对应的属性
Object	Number	getPropertiesForGID(gid)	通过指定GID查找其对应的属性

12.5.2　cc.TMXLayer 图层对象

cc.TMXLayer继承自cc.SpriteBatchNode，它是每个瓦片的父节点。瓦片集使用cc.TextureAtlas渲染，如果你在运行时直接或者间接地修改了瓦片的属性，那么这个瓦片将变成一个cc.Sprite，例如运行一个属性变化动作。默认情况下，不会有cc.Sprite对象被创建。不过，使瓦片变为cc.Sprite节点，便可以享受cc.Sprite丰富的API。

cc.TMXLayer最大的作用就是用来管理地图中的瓦片，它提供了一些管理瓦片的API，如表12-4所示。

表12-4　cc.TMXLayer管理瓦片的相关API

返回值类型	参数类型	函　数	说　明
cc.Sprite	cc.Point | Number Number	getTileAt(pos, y)	通过坐标获取对应的瓦片
Number	cc.Point | Number Number	getTileGIDAt(pos, y)	通过给定的瓦片坐标返回瓦片GID
Number	cc.Point | Number Number	getTileFlagsAt(pos, y)	通过给定的瓦片坐标返回瓦片标记
无	Number cc.Point | Number Number Number	setTileGID(gid, posOrX, flagsOrY, flags)	设置瓦片的GID
无	cc.Point | Number Number	removeTileAt(pos, y)	删除指定坐标上的瓦片
cc.Point	cc.Point | Number Number	getPositionAt(pos, y)	获取指定坐标位置（以点为单位）的瓦片坐标
Array	无	getTiles()	获取所有瓦片
无	Array	setTiles(tiles)	设置所有瓦片
cc.TMXTilesetInfo	无	getTileset()	获取层的瓦片设置信息
无	cc.TMXTilesetInfo	setTileset(info)	设置层的瓦片设置信息

如果你要隐藏obstacle层中位置为(2, 4)的瓦片，那么应该将指定位置的瓦片获取出来，然后设置它的不可见属性，代码如下：

```
1  loadTiledMap : function(){
2      var node = new cc.TMXTiledMap("res/unit12_tiledMap/TiledMap.tmx");
3      // ......
4      var layer = node.getLayer("obstacle");
5      var tile = layer.getTileAt(2, 4);
6      tile.setVisible(false);
7  }
```

上面的代码看上去似乎并没有什么问题，但实际上它是不安全的，例如当位置(2, 4)上没有瓦片的时候，就会报错。所以，若想要获取出一个瓦片的时候，应该先调用getTileGIDAt(pos, y)函数，判断当前位置是否有瓦片后再进行逻辑处理。要判断当前位置是否有瓦片，可通过getTileGIDAt(pos, y)函数的返回值进行辨别，如果返回的GID大于0，则表示当前位置上有瓦片存在，反之则没有，代码如下：

```
1  loadTiledMap : function(){
2      var node = new cc.TMXTiledMap("res/unit12_tiledMap/TiledMap.tmx");
3      // ......
4      var layer = node.getLayer("obstacle");
5      // 判断当前位置是否有瓦片
6      if (layer.getTileGIDAt(2, 4) > 0) {
7          var tile = layer.getTileAt(2, 4);
8          tile.setVisible(false);
9      }
10 }
```

另外，getPositionAt(pos, y)函数可以返回指定位置以点为单位的屏幕坐标，也可以理解为瓦片地图坐标转为Cocos2d-JS标准坐标。

而getTileset()函数可以返回当前层的一些信息，例如边距、瓦片图集的名字等。

此外，cc.TMXLayer还提供了一些大小、方向、名字等相关的API，这些API和cc.TMXTiledMap几乎是一致的，了解一下即可，具体如表12-5所示。

表12-5 cc.TMXLayer大小、方向、名字等相关的API

返回值类型	参数类型	函　数	说　明
cc.Size	无	getLayerSize()	获取层的大小
无	cc.Size	setLayerSize(size)	设置层的大小
cc.Size	无	getMapTileSize()	获取瓦片的大小
无	cc.Size	setMapTileSize(size)	设置瓦片的大小
Object	无	getProperties()	获取属性
无	Object	setProperties(properties)	设置属性
Number	无	getLayerOrientation()	获取层的方向（同地图方向）
无	Number	setLayerOrientation(orientation)	设置层的方向（同地图方向）
String	String	getProperty(propertyName)	通过属性名获取对应的属性
String	无	getLayerName()	获取层的名字
无	String	setLayerName(layerName)	设置层的名字

12.5.3　cc.TMXObjectGroup 对象组

对象组也称为对象层，这是因为Tiled Map Editor编辑器将对象组也放在图层当中，而Cocos2d-JS的API又将其称为组。

对象组中的对象不是场景中的节点，而是标记在当前地图层上的一些数据集合而已。标记有坐标属性，也可以有自定义属性，它们组合在一起，就可以满足多种数据配置的需求。

cc.TMXObjectGroup的API并不多，如表12-6所示。

表12-6　cc.TMXObjectGroup常用API

返回值类型	参数类型	函　数	说　明
Number	无	getPositionOffset()	获取子对象的偏移位置
无	Number	setPositionOffset(offset)	设置子对象的偏移位置
Object	无	getProperties()	获取存储在数组中的属性队列
无	Object	setProperties(properties)	设置存储在数组中的属性队列
String	无	getGroupName()	获取组名称
无	String	setGroupName(groupName)	设置组名称
Object	String	propertyNamed(propertyName)	通过属性名获取对应的属性
Object	String	getObject(objectName)	获取指定对象名的对象
Array	无	getObjects()	获取对象数组
无	Array	setObjects(objects)	设置对象数组

需要说明的是，cc.TMXObjectGroup中有一个_positionOffset属性，用来表示坐标偏移量，它只是一个简单的数据保存变量，对应的get和set函数是getPositionOffset()和setPositionOffset(offset)。所以，你可以将之前的代码改为如下的方式（注意第7行、第12行和第13行）：

```
1  loadTiledMap : function(){
2      var node = new cc.TMXTiledMap("res/unit12_tiledMap/TiledMap.tmx");
3      this.addChild(node);
4      node.setPosition((cc.winSize.width - node.width) / 2, 0);
5
6      var road = node.getObjectGroup("hero_road");
7      road.setPositionOffset(cc.p((cc.winSize.width - node.width) / 2, 0));
8      var objs = road.getObjects();
9      var roadX = null;
10     var pointArray = [];
11     for (var i = 0; i < objs.length; i++) {
12         //roadX=objs[i].x + (cc.winSize.width - node.width) / 2;
13         roadX = objs[i].x + road.getPositionOffset().x;
14         pointArray.push(cc.p(roadX , objs[i].y));
15     }
16 }
```

12.6　小结

通过本章的学习，我们知道了在Cocos2d-JS中游戏地图可以通过瓦片地图来解决。瓦片地图的原理是把图片中的元素进行单位化，将大图片拆成一张张小图块，然后通过组合拼接的方式，重新拼出整个地图。此外，我们也清楚了在开发中要使用Tiled Map Editor编辑器来制作瓦片地图。在Cocos2d-JS中，与瓦片地图有关的3个主要类是`cc.TMXTiledMap`、`cc.TMXLayer`以及`cc.TMXObjectGroup`。`cc.TMXTiledMap`中存放着对应的层，例如普通层和对象层（也称对象组）。普通层一般用来存放瓦片，而对象层则存放一些具有“标记性”功能的数据，例如你可以使用它来标记出怪物行走的道路、酷跑游戏中的花样金币组等。

12.7　参考资源

本章的参考资源如下所示。

- QT : http://www.qt.io/。
- TMXTiledMap API : http://api.cocos.com/cn/d6/d48/classcocos2d_1_1_t_m_x_tiled_map.html。

第三部分

高　级　篇

第 13 章

反射调用

13

在Native上，可能会获取移动设备的一些信息，例如获取手机系统版本、UUID（手机串号）、网络情况等，这些在不同平台上的实现方式是不一样的。在iOS中，我们可以通过Objective-C代码获取；在Android上，我们可以通过Java代码实现。但是，我们的逻辑代码一般都用JavaScript去编写，那如何在JavaScript层获取到移动设备的信息呢？或许可以通过Cocos2d-JS 3.x之后提供的反射调用机制来实现。

本章内容：

- 反射调用概述
- 反射调用Objective-C
- 反射调用Java

13.1 反射调用概述

在Cocos2d-JS 3.0 beta版本中，引擎加入了一个新特性——反射调用。在iOS和Android平台上，我们可以在JavaScript中通过反射直接调用Objective-C和Java类中的静态方法。但是不管是反射调用Objective-C，还是反射调用Java，其函数接口都是一致的，代码如下：

```
1  // Objective-C反射调用接口
2  var ret = jsb.reflection.callStaticMethod(className, methodName, arg1, arg2, ...);
3  // Java反射调用接口
4  var ret = jsb.reflection.callStaticMethod(className, methodName, methodSignature,
   parameters, ...);
```

可以明显看到，虽然反射调用Objective-C和反射调用Java的函数接口一致，但其形参似乎有些不同。这是因为反射调用Objective-C时，只需传入要调用的类名、方法名、参数即可，而在反射调用Java时，我们还需要对方法进行签名。

> **注意** 不管是到Objective-C的反射，还是到Java的反射，都仅支持其类的静态方法。

13.2 反射调用 Objective-C

反射调用Objective-C较为简单，只需传入要调用的Objective-C类名、方法名、参数即可。假设现在有一个叫Utils的类，它实现了获取当前设备类型的静态方法，其代码如下：

Utils.h

```
1  #import <Foundation/Foundation.h>
2  @interface Utils : NSObject
3  // 获取[移动设备类型]
4  + (NSString *)getDeviceModel;
5  @end
```

Utils.m

```
1  #import "Utils.h"
2  @implementation Utils
3  + (NSString *)getDeviceModel {
4      NSString *deviceModel = [[UIDevice currentDevice] model];
5      return deviceModel;
6  }
7  @end
```

上述代码中的getDeviceModel函数获取了设备的类型，其返回值可能是iPhone、iPad等。在JavaScript中，可以通过如下代码反射调用Utils类中的getDeviceModel函数：

```
1  if (cc.sys.isNative && cc.sys.os == cc.sys.OS_IOS) {
2      var ret = jsb.reflection.callStaticMethod("Utils", "getDeviceModel");
3      cc.log(ret);
4  }
```

值得注意的是，对于上述代码的条件判断，我们必须保证当前运行的环境是iOS系统以及在Native上。这是因为在iOS浏览器上，cc.sys.os的值也为cc.sys.OS_IOS，但此时并非运行在JSB环境上，所以调用jsb.reflection.callStaticMethod()函数会报错。

另外，jsb.reflection.callStaticMethod()函数中的形参以及返回值都仅支持int、float、bool、string类型，其他类型暂不支持。

说明 反射调用Objective-C时，参数className为类名即可，无需包含路径。

13

13.3 反射调用 Java

相对反射调用Objective-C而言，反射调用Java显得复杂一些，一方面，反射调用Java中的className参数必须是包含Java包路径的完整类名。另一方面，反射调用Java时必须对方法进行签名。

如图13-1所示，在org.cocos2dx.javascript包中有一个NativeJavaClass类，类中有一个返回值是String类型的静态方法getHelloWorld，那么通过jsb.reflection.

callStaticMethod(className, methodName, methodSignature, parameters, …)反射调用getHelloWorld方法的代码应该是这样的：

```
1  // 判断是否是Android中的Native环境
2  if (cc.sys.isNative && cc.sys.os == cc.sys.OS_ANDROID) {
3      var ret = jsb.reflection.callStaticMethod("org/cocos2dx/javascript/
       NativeJavaClass", "getHelloWorld", "()Ljava/lang/String;");
4      cc.log("From JBS, Result is :" + ret);
5  }
```

其中，className的完整类名为org/cocos2dx/javascript/NativeJavaClass。值得注意的是，这里是代码文件的路径，而非Java中的包名，所以采用的是斜杠/。getHelloWorld对应NativeJavaClass中的静态方法getHelloWorld，而()Ljava/lang/String;为方法签名，此签名表示该反射调用函数没有参数，但返回值是String类型。方法签名是反射调用Java所特有的一个参数，其类型是一个字符串。下面给出了NativeTavaClass类的定义：

```
1  package org.cocos2dx.javascript;
2  public class NativeJavaClass {
3      public static String getHelloWorld(){
4          return "Hello World";
5      }
6  }
```



图13-1　JSB_Java工程结构

方法签名的格式是：(PP)R。P为parameter，表示参数，支持0到多个参数，R为result，表示返回值。最简单的方法签名是：( )V，它表示一个没有参数没有返回值的方法。

目前，Cocos2d-JS中支持Java数据类型的签名有4种，具体见表13-1所示。

表13-1　Cocos2d-JS中支持的Java数据类型

Java数据类型	签　　名
int	I
float	F
boolean	Z
String	Ljava/lang/String;

表13-2列举一些签名例子。

表13-2　签名示例

签　名	说　明
(F)V	一个参数，参数类型为float类型，没有返回值
(IF)Ljava/lang/String;	两个参数，第一个参数为int类型，第二个参数为float类型，返回值为String类型
(II)Z	两个参数，两个参数都为int类型，返回值是boolean类型

注意　String类型签名的末尾还有一个分号（;）。

13.4　小结

通过本章的学习，我们知道了如何反射调用Objective-C和Java类的静态方法，这在Native上是非常有用的。例如，我们可以通过反射调用获取到手机的UUID，将其生成为游客账号。通过反射调用，判断设备是否处于Wi-Fi网络环境下。此外，也可以通过反射调用来接入广告、统计以及支付等第三方SDK。

13.5　参考资源

本章的参考资源为http://cocos2d-x.org/docs/manual/framework/cocos2d-js/catalog/index.html。

第 14 章

Chipmunk物理引擎

在游戏中，我们经常需要去模仿真实场景的物理行为，比如《愤怒的小鸟》抛物线的弹道，汽车的碰撞等，如果我们通过编码来实现这些物理模拟，将会涉及许多数学和物理公式，实现起来既复杂又容易出错，这时就需要用到物理引擎啦。

物理引擎可以给物体赋予物理属性，使物体产生运动、碰撞以及旋转，从而模拟真实世界物体的运动。它可以让我们的游戏世界变得更加真实。

Cocos2d-JS内嵌了两种2D物理引擎，分别是Box2D和Chipmunk，它们的概念大同小异。但是因为Cocos2d-JS目前在JSB上仅支持Chipmunk物理引擎，所以本章只讲解该物理引擎。

本章内容：

- 物理概念
- Chipmunk中的基本概念
- 刚体
- 形状
- 约束
- 空间
- 碰撞检测
- 查询
- 实例——拖动刚体的实现

14.1 物理概念

首先，我们来学习一些物理概念，以及它们在Chipmunk上是如何表现的。

14.1.1 刚体

刚体（rigid body）是指在运动中或者受力后形状和大小不变，以及内部点的相对位置都不会改变的物体。在现实世界中，刚体是不存在的，因为无论再坚硬的物体，受到力的作用后都会产生形变，然而对于程序而言，刚体的特性是最容易模拟和实现的。

14.1.2 面积

面积（area）是指物体所围成的平面图形的大小。对于2D物理引擎来说，你可以简单地把它理解为物体的体积。在Chipmunk中，面积的标量跟长宽等其他标量一致，例如一个10×20的长方体，它的面积就是200。

14.1.3 质量

质量（mass）指的是物体中所含物质的量，即物体惯性的大小。通俗地说，即我们平常说的物体的重量。在Chipmunk中，质量的值由自己指定，它只是一个相对的量。比较推荐的做法是，跟物体的密度相结合，采用易于计算的方式来估算质量的值。比如，一个面积为200的物体，估算的密度为2，那么它的质量即是200×2=400。

14.1.4 密度

密度（density）指的是物体单位体积的质量大小，即“密度=质量/体积”，但对于2D物理引擎来说，并不存在体积的概念，所以你需要把这里的体积理解成面积。Chipmunk不会要求你去显式设定物体的密度，它将隐式地由面积和质量决定，但是在编码的过程中，如果我们把密度作为一个标量去设定各个物体的质量值，将有助于我们控制游戏的物理表现。

14.1.5 转动惯量

转动惯量（moment of inertia）是物体绕轴转动时的惯性的量度，它取决于物体的质量、形状以及转轴的位置，这个值通常难以估算，所以你可以通过Chipmunk提供的几个计算转动惯量的函数来计算它。

14.1.6 弹性系数

弹性系数（elasticity）是材料在弹性极限内应力与应变之比，它决定了物体产生碰撞后所受到的反弹力的大小。在Chipmunk中，我们可以用很直观的方式来设定它：0.0为无反弹，1.0为完美反弹。当两物体相互碰撞时，它们的弹性系数将由它们俩的弹性系数的乘积来决定。

14.1.7 摩擦系数

摩擦系数（friction）是指两表面间的摩擦力和作用在其表面上的垂直力的比值。Chipmunk 中的摩擦采用的是库仑摩擦力（Coulomb’s Friction），0.0为无摩擦。如果你需要精确地控制摩擦系数，可以去查阅下库仑摩擦模型的相关定义（见本章参考资源）。

14.2　Chipmunk 中的基本概念

了解了物理上的一些概念之后，再来看看Chipmunk中的基本概念。

14.2.1　基本类型

在Chipmunk中，你将接触到4种类型，具体如下所示。

- **刚体（rigid body）**：刚体保存了物体的物理属性，例如质量（mass）、位置（position）、旋转角度（rotation）、速度（velocity）等。你可以将它附着到Sprite上来使你的Sprite拥有物理属性。
- **形状（shape）**：形状赋予了物体碰撞体积，使物体可以参与碰撞。你可以将形状附着到刚体上来使用它。一个刚体可以附着零到任意多个形状。形状包含了物体表面的摩擦系数（friction）以及弹性系数（elasticity）等属性。
- **约束/关节（constraint/joint）**：约束和关节描述了物体之间相互间的附着关系以及它们将如何产生相互作用，例如手和身体间的连接，或者汽车的车身和车轮间的连接，均可以使用关节来连接。（关节也属于约束，后面我们将以约束来代替所有的“约束/关节”。）
- **空间（space）**：空间是模拟真实物理世界的场所，你可以将刚体、形状和关节加入空间，使其控制和模拟它们之间的相互作用。

14.2.2　Chipmunk 的坐标系统

Chipmunk拥有自己的一套坐标系统，它独立于Cocos2d-JS，所以我们需要依据自己的需求跟Cocos2d-JS中的坐标系相结合。

Chipmunk的坐标系统很简单，它是一个二维笛卡儿坐标系，虽然独立于Cocos2d-JS的坐标系，但是也和Cocos2d-JS坐标系有所类似，也都分为X轴和Y轴。其中，水平向右表示X轴的正方向，竖直向上为Y轴的正方向，而左下角同样为坐标系的原点。我们可以很便利地将它等比例地映射到Cocos2d-JS当中去，可以通过改变它跟Cocos2d-JS中坐标系的相对比例关系来对它进行缩放，或者也可以跟某个层的坐标系相结合，通过改变层的大小来达到缩放的效果。但是需要注意的是，我们无法对Chipmunk中局部坐标系统进行缩放，所以无法像Cocos2d-JS那样通过缩放子坐标系的方法来缩放某一部分的物体。

在Chipmunk中，我们还需要用到一个坐标系，就是相对于刚体位置的一个相对坐标系。它只是世界坐标系相对于刚体位置的一个位移转换，即是一个以刚体位置为坐标原点，以与世界坐标系同比例构成的一个二维笛卡儿坐标系。同时，Chipmunk也提供了几个便利的函数来供我们做相应的坐标系转换。

14.2.3　Chipmunk 中的基本数据结构

Chipmunk中的基本数据结构有`cp.Vect`和`cp.BB`，下面一起来看一下。

1. cp.Vect

它的概念同Cocos2d-JS中的cc.Vec2，即为二维坐标系的一个点或向量，你可以通过cp.v(x, y)函数来创建它。同时你可以使用cp.vzero来代表原点(0, 0)，注意不要去改变cp.vzero的值。

与cp.Vect相关的操作如表14-1所示。下面所列的变量vect、参数v、v1以及v2均为cp.Vect的实例，cp.Vect的实例对象调用方法将改变自身的值，而调用cp.v.func()格式的函数将返回计算结果。

表14-1 与cp.Vect相关的操作

对应的属性或方法	说 明
vect.**x**	X轴坐标值
vect.**y**	Y轴坐标值
vect.add(v2)	同cp.v.add(v1, v2)，使两个向量相加（v1+v2）
vect.sub(v2)	同cp.v.sub(v2)， 使两个向量相减（v1-v2）
vect.mult(v2)	同cp.v.mult(v, s)，使向量与数相乘（v*s）
vect.neg()	同cp.v.net(v)，取反向量（-v）
cp.v.dot(v1, v2)	计算两个向量的点积
cp.v.len(v)	计算向量的长度
cp.v.len2(x, y)	同上，分别传入x和y的坐标
cp.v.eql(v1, v2)	计算两向量是否相等
cp.v.cross(v1, v2)	计算两向量的叉积，注意返回的结果为Z轴坐标的长度
cp.v.perp(v)	计算向量的垂直向量，即将向量顺时针旋转90度
cp.v.pvrperp(v)	将向量逆时针旋转90度
vect.project(v2)	同cp.v.project(v1, v2)，计算v1向量在v2向量上的投影向量
vect.rotate(v2)	同cp.v.rotate(v1, v2)，使用复数乘法使v1沿v2做旋转，如果v1不是单位向量，则会被拉伸
cp.v.unrotate(v1, v2)	反转cp.v.rotate
cp.v.lengthsq(v)	计算向量的长度的平方，如果你只需要比较长度，此方法的速度比cp.v.len快
cp.v.lengthsq2(x, y)	同上，分别传入x和y的坐标
cp.v.normalize(v)	计算v的单位向量
cp.v.normalize_safe(v)	同上，但遇到零向量时返回零向量而非产生除零错误
cp.v.clamp(v, len)	限制v的长度在len以内
cp.v.lerp(v1, v2, t)	以t百分比（小数表示）在v1与v2间做线性插值
cp.v.lerpconst(v1, v2, d)	以不超过d的长度在v1与v2间做线性插值
cp.v.slerp(v1, v2, t)	以t百分比（小数表示）在v1与v2间做球形线性插值
cp.v.slerpconst(v1, v2, t)	以不超过t的弧度值在v1与v2间做球形线性插值
cp.v.dist(v1,v2)	计算v1与v2的距离
cp.v.distsq(v1, v2)	计算v1与v2的距离的开方，如果你只需要比较长度，此方法的速度比cp.v.dist快
cp.v.near(v1, v2, dist)	当v1跟v2的距离比dist小时，返回true
cp.v.forangle(a)	返回以a作为弧度值的单位向量

（续）

对应的属性或方法	说　　明
cp.v.toangle(v)	返回以v向量作为方向的弧度值
cp.v.str(v)	返回string表示的向量，做debug用

2. cp.BB

cp.BB 类型主要用于物理的碰撞检测，称为轴对齐包围盒，又称AABB包围盒，如图14-1所示。它类似于Cocos2d-JS的cc.Rect，但又有所不同。cc.Rect表示一个区域，其成员属性为x、y、width、height，分别表示这个区域的X轴坐标、Y轴坐标、宽度以及高度。而cp.BB表示一面（top）。实际上，cp.BB在l = x、b = y、r = x + width、 t = y + height时所表个包围盒，其成员属性为l、b、r、t，分别表示左边（left）、下面（bottom）、右边（right）、上示的区域范围与cc.rect(x, y, width, height)一致。

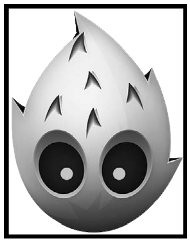


图14-1　AABB包围盒

包围盒只是Chipmunk内部使用的数据类型，通常情况下不需要使用它，不过理解它对于我们理解Chipmunk的碰撞检测原理会有所帮助。物理引擎常用的3种包围盒示例：球体包围盒、轴对齐包围盒（AABB）、有向包围盒（OBB）。在Chipmunk中，只使用了轴对齐包围盒。

14.3　刚体

在Chipmunk中存在两种刚体：动态刚体（dynamic body）和静态刚体（static body）。

动态刚体是默认的刚体类型，它们会受力、重力和碰撞的影响，拥有有限的质量，并且会产生碰撞回调，是你希望物理引擎模拟的刚体类型。

静态刚体是基本上不移动的刚体类型，你可以很方便地用它来模拟地面、墙体等不受重力影响的物体。

在学习刚体相关的API之前，我先创建出Chipmunk的物理空间（或称为物理世界），这将在一定程度上帮助你学习后面的知识。就像我在2.8节中所说的那样，在学习相关知识点的时候，我们应该立刻结合代码动手操作，加强实战，最后从代码运行的结果便可得出学习成果，从而降低了学习成本，提高了学习效率。

先创建出一个最基本的物理空间，并稍微讲解一下代码，此时你可能对下面的代码不是很熟悉，但是无妨，先跟着做即可，你可以在阅读完14.6节之后再回过头来细看这些代码。

说明 使用Chipmunk物理引擎时，需要在project.json的`modules`数组中添加`chipmunk`，如下所示：

```
1 "modules" : ["cocos2d", "chipmunk"],
```

下列代码用于创建出物理空间：

```
1  // 加载[物理空间]
2  loadPhysicsSpace : function(){
3      this.space = new cp.Space();              // 创建[物理空间]
4      this.space.gravity = cp.v(0, -150);       // 设置[物理空间重力]
5      this.space.sleepTimeThreshold = 0.5;      // 设置[睡眠检测临界值]
6
7      var thickness = 100;     // 物理空间围墙厚度
8      var staticBody = this.space.staticBody; // 创建[静态刚体]
9      var walls = [
10         new cp.SegmentShape(staticBody, cp.v(-thickness, -thickness),
           cp.v(cc.winSize.width, -thickness), thickness),  // 底部
11         new cp.SegmentShape(staticBody, cp.v(-thickness, -thickness),
           cp.v(-thickness, cc.winSize.height), thickness), // 左边
12         new cp.SegmentShape(staticBody, cp.v(cc.winSize.width + thickness,
           -thickness), cp.v(cc.winSize.width + thickness, cc.winSize.height),
           thickness ),      // 右边
13         new cp.SegmentShape(staticBody, cp.v(-thickness, cc.winSize.height +
           thickness), cp.v(cc.winSize.width, cc.winSize.height + thickness),
           thickness)       // 顶部
14     ];
15
16     // 创建围墙
17     for(var i = 0; i < walls.length; i++){
18         var shape = walls[i];
19         shape.setElasticity(0.5);    // 设置[围墙的弹性系数]
20         shape.setFriction(0);        // 设置[围墙的摩擦力]
21         this.space.addStaticShape(shape); // 添加[围墙到物理空间中]
22     }
23 }
```

第9行到第14行描述了物理空间4个边的形状，它们相当于4块木板，固定出一个矩形空间，而这个空间就是物体运动的场所。实际上，物理空间所表现的一切物理信息都是不见的，例如空间的大小、物体的形状等。但是，Cocos2d-JS提供了一个`PhysicsDebugNode`函数，它可以将整个物理空间绘制出来，代码如下：

```
1  // 加载[物理空间调试节点]
2  loadDebugPhysicsSpace : function(){
3      var node = new cc.PhysicsDebugNode(this.space);
4      this.addChild(node, Number.MAX_VALUE);
5  }
```

此时运行代码，可以看到构建出来的物理空间边界。但需要说明的是，因为在前面的代码中，我设定的物理空间的边界正好是设备屏幕的边界，所以，应当将当前的Layer进行一定的缩放，才可以看到物理空间边界的DebugNode，图14-2为当前Layer缩放8/10之后所表现出来的物理空间边界。



图14-2　物理空间Debug

14.3.1　动态刚体

创建好物理空间之后，你便可以通过如下方式来创建动态刚体：

```
1  var body = new cp.Body(mass, moment);
```

动态刚体的创建需要两个参数，分别是质量（mass）和转动惯量（moment of inertia），质量决定了刚体移动时的惯性，而转动惯量决定了刚体转动时的惯性。通常，我们很容易决定一个物体的质量，却很难去决定一个物体的转动惯量。幸运的是，Chipmunk给予了我们如下几个便利的函数来计算刚体的转动惯量：

```
1  cp.momentForCircle(mass, radius1, radius2, offset)
2  cp.momentForSegment(mass, a, b, radius)
3  cp.momentForPoly(mass, verts, offset)
4  cp.momentForBox(mass, width, height)
5  cp.momentForBox2(mass, box)
```

如果你为刚体添加了多个形状（shape），就可以通过上述函数分别计算它们的转动惯量并把结果相加，从而得到想要的转动惯量。

你也能够很轻松地通过如下接口去动态获取和设置物体的质量和转动惯量（body为cp.Body的实例，下同）：

```
1  var mass = body.m;        // [获取]质量
2  body.setMass(mass);       // [设置]质量
3  var moment = body.i;      // [获取]转动惯量
4  body.setMoment(moment);   // [设置]转动惯量
```

同时Chipmunk还提供了如下几个便利的函数来计算刚体的面积，以便我们去估算刚体的质

量、密度等属性：

```
1  // 计算空心圆的面积，r1为外圆半径，r2为内圆半径，r2等于0时为实心圆面积
2  cp.areaForCircle(r1, r2)
3  // 计算线段面积，a和b为起点和终点，r为线段的宽度
4  cp.areaForSegment(a, b, r)
5  // 计算凸多边形的面积，verts为cp.Vect类型的Array
6  cp.areaForPoly(verts)
```

下面我在当前的物理空间中创建出一个动态刚体，相关代码如下：

```
1  // 加载[精灵刚体]
2  loadSpriteBody : function(){
3      // 创建物理精灵
4      var node = new cc.PhysicsSprite(res.sh_node_256_png);
5      this.addChild(node);
6
7      var mess = 10;      // 质量
8      var body = new cp.Body(mess, cp.momentForBox(mess, node.width, node.height));
9      this.space.addBody(body);
10     body.setPos(cp.v(cc.winSize.width /2, cc.winSize.height /2));
11
12     var shape = new cp.BoxShape(body, node.width, node.height);
13     this.space.addShape(shape); // 为物理空间添加形状
14     node.setBody(body);         // 精灵设置刚体
15     shape.setElasticity(0.5);   // 设置[弹性系数]
16     shape.setFriction(0.5);     // 设置[摩擦力]
17 }
```

第4行代码用于创建一个物理精灵，cc.PhysicsSprite继承自cc.Sprite，并且提供了可以设置刚体的API，如第14行代码将cc.PhysicsSprite和cp.Body绑定在一起，使得精灵拥有了刚体的物理特性。运行代码后，效果如图14-3所示。需要说明的是，cc.PhysicsDebugNode给动态刚体绘制出的Debug信息区域为红色的，而静态刚体则为灰色的。

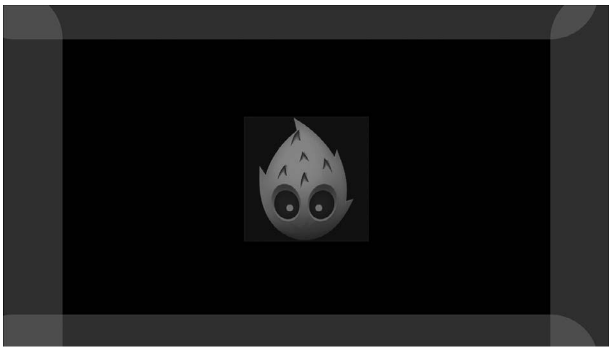


图14-3 物理空间和物理精灵

除此之外，cp.Body提供了一些接口，具体如下：

```
1  body.getPos()          // [获取]body的位置
2  body.setPos(pos)       // [设置]body的位置
```

```
3  body.a                   // [获取]body的角度
4  body.setAngle(angle)     // [设置]body的角度
5  body.getVel()            // [获取]body的速度
6  body.setVel(vel)         // [设置]body的速度
7  body.getAngVel()         // [获取]body的转速
8  body.setAngVel(vel)      // [设置]body的转速
```

此外，还可以通过以下几个接口给刚体施加力：

```
1  var force = body.f                // [获取]施加在刚体上的力
2  body.resetForces()                // [重置]施加在刚体上的力
3  body.applyForce(force, r)         // 在刚体上施加[力]
4  body.applyImpulse(impulse, r)     // 在刚体上施加[冲量]
```

其中，force和impulse都为cp.Vect类型，r为力（或冲量）的作用点离刚体位置的偏移量。另外，需要注意的是力（force）和冲量（impulse）的区别，力指的是持续的力，将会一直作用在刚体上，例如推箱子的力。而冲量指的是瞬间的力，只有在函数调用时，才给刚体施加一次瞬间的冲击力，例如高尔夫球杆击打球的力。现在，我给之前的刚体添加一个冲量，代码如下：

```
1  body.applyImpulse(cp.v(3000, 3000), cp.v(0, 0));
```

另外，可以通过以下接口给刚体添加和移除形状：

```
1  body.addShape(shape)        // [添加]形状
2  body.removeShape(shape)     // [移除]形状
```

也可以通过以下接口移除刚体上的约束：

```
1  body.removeConstraint(constraint) // [移除]约束
```

同时，cp.Body还提供了几个便利的函数来遍历刚体上的形状、约束和碰撞对（关于碰撞对（collision pairs）的内容，将在后续章节中介绍）：

```
1  body.eachShape(func)
2  body.eachConstraint(func)
3  body.eachArbiter(func)
```

其中，func为回调函数，每个func都携带一个参数，分别代表了形状、约束以及碰撞对，它在body上的每个形状、约束和碰撞对上均会执行一次回调函数。例如，你要遍历出刚体所有的形状，那么代码应该如下：

```
1  var func = function(shape){
2      cc.log(shape);
3  };
4  body.eachShape(func);
```

其中函数变量func的参数shape即代表形状。

另外，值得注意的是，使用eachArbiter函数前，需要开启space.enableContactGraph（具体请参照14.6节）。

有时候，我们需要在body和space之间的坐标系统进行相互转换，这类似于Cocos2d-JS坐标

系中本地坐标和世界坐标的概念。下面这两个函数将给我们带来这样的便利：

```
1  body.local2World(vect)     // 将相对于body的坐标转换为世界坐标
2  body.world2Local(vect)     // 将世界坐标转换为相对于body的坐标
```

下面我们来看一下cp.Body提供的几个状态查询函数:

```
1  body.isSleeping()    // 查询body是否在sleep的状态
2  body.isStatic()      // 查询body是否为静态刚体
3  body.isRogue()       // 查询body是否被加入到空间中，未加入将返回true
```

睡眠（sleep）是这样一种机制：当Chipmunk检测到刚体在一段时间内都处于静止状态时，它将会使刚体进入sleep状态，直到它受到新的力或者碰撞或者被唤醒。对于已经进入sleep状态的刚体，Chipmunk将不用关心和处理它们的某些状态，从而节省了CPU的时间和电池的寿命。

cp.Body有如下几个和sleep相关的接口：

```
1  body.activate()               // 激活刚体
2  body.activateStatic()         // 激活静态刚体
3  body.sleep()                  // 使刚体睡眠
4  body.sleepWithGroup(group)    // 使刚体随着组睡眠
```

注意，body.sleepWithGroup(group)中的参数group为cp.Body类型，并非碰撞检测中的组。当Chipmunk中的刚体睡眠时，它将会使所有碰触或连接的刚体睡眠，我们称其为组。当组中的一个刚体被激活时，全组刚体将被激活。此接口允许你设置刚体随着一组刚体睡眠，当此组被激活时，刚体将被激活。

14.3.2 静态刚体

静态刚体的创建方式很简单，可以使用如下接口创建：

```
1  var staticBody = new cp.StaticBody();
```

实际上，staticBody也是cp.Body的实例，所以你可以使用cp.Body上所有的接口，也可以通过如下方式来创建静态刚体:

```
1  var staticBody = new cp.Body(Infinity, Infinity); //Infinity值为正无穷
```

但是我们并不推荐采用此种方式来创建静态刚体，因为它通常会使与其接触的刚体无法进入sleep状态而造成性能损失。如果我们用此方式来创建静态刚体，Chipmunk往往并不认为它属于静态刚体，调用body.isStatic()函数将返回false，所以它并不像其他静态刚体那样始终处于sleep状态，导致Chipmunk很难去决定是否要将与其接触的其他物体设置为sleep状态，从而造成不必要的性能损失。

当然，这种方法并不是一无是处。由于静态刚体的特殊性，当我们为静态刚体添加形状并把形状加入到空间后，形状的位置将无法随着静态刚体位置的改变而改变。也就是说，我们将无法再移动这些形状。然而如果我们用上述方法创建静态刚体，将不会遇到此类问题，有时候这会是一个很有用的功能，比如我们需要创建一堵可以移动的墙。所以，当需要一个静态刚体时，请根

据项目需求慎重做出选择。

另外，由于静态刚体不可移动的特性，在一个空间中，你往往只需要一个静态刚体就足够了，我们需要做的只是把所有不可移动的形状添加到这个静态刚体中。所以，Chipmunk已经为我们提供了这样一个静态刚体，我们可以选择`space.staticBody`来使用它，而不用自己再去创建了。下列代码创建出一个静态刚体：

```
1  // 加载[静态刚体]
2  loadStaticBody : function(){
3      // 创建物理精灵
4      var node = new cc.PhysicsSprite(res.sh_node_256_png);
5      this.addChild(node);
6
7      // 创建静态刚体
8      var body = this.space.staticBody;
9      node.setBody(body); // 精灵绑定刚体
10     body.setPos(cp.v(cc.winSize.width/ 2, cc.winSize.height /2));
11
12     // 形状创建
13     var shape = new cp.BoxShape(body, node.width, node.height);
14     this.space.addShape(shape); // 将形状添加到物理空间
15     shape.setElasticity(1);     // 设置[弹性系数]
16     shape.setFriction(1);       // 设置[摩擦力]
17 }
```

14.4　形状

在现实世界中，一个物体仅拥有`cp.Body`里面的那些属性是不够的，还需要一个重要的属性：形状。所以，Chipmunk也提供了这样一个类来供我们模拟现实世界：`cp.Shape`。

在现实世界中，物体往往会发生形变，比如吹弹可破的肌肤、摔在地上就会碎裂的玻璃杯、汽车开过就会溅起的小水滩，然而Chipmunk的形状并不能满足这些需求，因为Chipmunk只能模拟刚体的行为。但是，这并不是说我们的程序就无法实现上述这些效果，我们照样可以使用一些很精妙的技巧去实现它。比如，可以通过移除一整个刚体使用许多较小而又分散的刚体来代替它，并使这些刚体碎片拥有不同方向的初速度来模拟碎裂的效果。再比如，可以把许多非常小的刚体使用约束连接在一起，从而形成一个大的物体来模拟该物体在受力时产生的形变效果等。当然，这就需要各位读者充分发挥自己的想象力啦。另外，Google的LiquidFun流体引擎也可实现流体物理特性的效果，读者若有兴趣，可去http://google.github.io/liquidfun/查阅相关的资料。

下面我们就来学习一下Chipmunk中的形状。

14.4.1　基类 `cp.Shape`

在Chipmunk中，存在这样3种基本的形状。

- 圆（circle）：即圆形。

- 线段（line segment）：它通常用来做墙、地面、边界等，其宽度能够设置。它一般是添加在静态刚体上，但我们也可以给它设置较大的宽度，把它作为两头为半圆形的长条形状来看待，如图14-1所示。
- 凸多边形（convex polygon）：注意，Chipmunk仅支持凸多边形，如果你需要凹多边形，可以通过组合多个凸多边形来实现。我们还可以通过以下接口来创建四边形：

```
1  // width和height分别表示宽和高，为Number类型
2  var box = cp.BoxShape(body, width, height);
3  // box为cp.BB类型
4  var box2 = cp.BoxShape(body, box);
```

  实际上，这两个BoxShape都属于cp.PolyShape类的实例，这两个接口只是能够方便地创建四边形而已，所以它们能够使用cp.PolyShape的所有方法。

通过以上几个形状，我们就能够组合出各种各样的形状来，同时，它们都继承自cp.Shape类，所以下面就来关注cp.Shape都拥有哪些属性（变量shape为cp.Shape的实例，下同）。

下面的代码用于设置和获取弹性系数以及摩擦力：

```
1  var elasticity = shape.e;    // 获取[弹性系数]
2  shape.setElasticity(e);      // 设置[弹性系数]
3  var friction = shape.u;      // 获取[摩擦力]
4  shape.setFriction(u);        // 设置[摩擦力]
```

而碰撞检测相关的参数将在后续章节中进一步介绍：

```
1  var group = shape.group;     // 获取[碰撞检测相关的组]
2  group = 10;                  // 设置[碰撞检测相关的组]
3  var layers = shape.layers;   // 获取[碰撞检测相关的层]
4  layer.setLayers(layers);     // 设置[碰撞检测相关的层]
```

传感器相关接口的值为Boolean类型相关代码如下：

```
1  var isSensor = shape.sensor;// 获取[是否只是传感器]的结果
2  shape.setSensor(isSensor);  // 设置[是否只是传感器]
```

若shape.sensor的值为true，则只调用碰撞回调，并不会产生真实的碰撞。更多碰撞检测相关的内容请查看14.7节。

获取和设置刚体时，需要注意的是，只能在shape未加入space时才能设置刚体，否则会报错。设置和获取刚体的代码如下：


```
1  var body = shape.getBody(); // 获取[刚体]
2  shape.setBody(body);        // 设置[刚体]
```

获取AABB包围盒的代码如下：

```
1  shape.getBB();                // 获取[包围盒]
```

更新位置和旋转角度的代码如下，其中pos和rot均为cp.Vect类型：

```
1  shape.update(pos, rot);
```

此外，cp.Shape还有一些查询相关的接口（将在14.8节中进一步讲解），具体如下：

```
1  // 点查询，p为cp.Vect类型
2  shape.pointQuery(p)
3  // 最近点查询，p为点坐标，是cp.Vect类型
4  shape.nearestPointQuery(p)
5  // 线段查询，其中a和b为线段的起点和终点，它们是cp.Vect类型
6  shape.segmentQuery(a, b)
```

14.4.2　圆形

圆形（circle shape）的创建方式如下：

```
1  var circle = new cp.CircleShape(body, radius, offset);
```

其中radius为Number类型，表示圆的半径。offset是cp.Vect类型，为圆形与刚体重心的偏离值。可以通过如下方式来获取圆形的相关属性（注意不要改变它们，其中circle为cp.CircleShape的实例）：

```
1  var radius = circle.r   // 获取[半径]
2  var offset = circle.c   // 获取[偏离值]
```

14.4.3　线段

线段（line segment）的创建方式如下：

```
1  var segment = new cp.SegmentShape(body, a, b, r);
```

其中，a和b均为cp.Vect类型，表示线段的起点和终点，r为数字类型，是线段的宽度。同时，可以通过如下方式来获取线段的相关属性（注意不要改变它们，其中segment为cp.SegmentShape的实例）：

```
1  var a = segment.a   // 获取[线段起点]
2  var b = segment.b   // 获取[线段终点]
3  var r = segment.r   // 获取[线段厚度]
4  var n = segment.n   // 获取[线段法线]
```

此外，还可以重新设置线段的位置：

```
1  segment.setEndpoints(a, b);
```

其中a和b同样均为cp.Vect类型，表示线段的起点和终点。另外值得注意的是，当我们把几条线段连接起来，它们往往直接会因为产生碰撞而出现缝隙，如果想要避免这种情况，需要用到如下接口：

```
1  segment.setNeighbors(prev, next);
```

其中，prev和next均为cp.Vect类型，分别表示上一条和下一条线段的终点。

14.4.4　凸多边形

凸多边形（convex polygon）的创建方式如下：

```
1 var poly = new cp.PolyShape(body, verts, offset);
```

其中verts为Array类型，表示凸多边形点（cp.Vect）的集合，offset为cp.Vect类型，表示与刚体重心的偏离值。同时，我们可以通过如下方式来获取和设置凸多边形的相关属性（注意不要改变它们，其中poly为cp.PolyShape的实例）：

```
1 // 获取[凸多边形的顶点集合]
2 var verts = poly.verts;
3 // 设置[凸多边形的顶点集合]，其参数含义同cp.PolyShape
4 poly.setVerts(verts, offset);
```

14.5 约束

在Chipmunk中，约束（constraint）都是基于速度的约束，这意味着它们主要通过同步两个刚体的速度进行作用。约束将速度视为其主要的输入并产生一个速度的变化来作为它的输出。一些约束（尤其是关节）通过改变速度来修正位置的差异。

严格来说，两个刚体的弹簧并不是约束，它会创建一个力来影响两个刚体的速度，弹簧关联的阻尼才是真正的约束，但是我们通常并不需要关心这些，但还是会将弹簧作为约束的一部分来看待。

14.5.1 基类 cp.Constraint

Chipmunk中所有的约束都继承自cp.Constraint。下面先来关注一下它都有哪些属性，其中constraint为cp.Constraint的实例对象：

```
1 // 连接的第一个刚体（cp.Body）
2 constraint.a;
3 // 连接的第二个刚体（cp.Body）
4 constraint.b;
5 // 两物体间可用的最大作用力，为数字类型，默认为Infinity，即正无穷
6 constraint.maxForce;
7 // 一秒后保留的误差百分比，默认为Math.pow(1-0.1, 60)，即每1/60秒修正10%的误差
8 constraint.errorBias;
9 // 约束能够实施错误修正的最大速度，默认为Infinity，即正无穷
10 constraint.maxBias;
```

Chipmunk总共提供了10种约束来供我们使用，下面我将会逐一介绍它们。

说明 以下约束名称均没有做中文翻译，原因有二，其一是因为大多数约束都没有浅显易懂的中文翻译名称，反比英文名称来得晦涩难懂。其二，英文名称跟类名称一一对应，平时使用起来也比中文名称更能明白其所指，所以本书直接使用了约束的英文名称而未做翻译。

14.5.2　Pin Joint

Pin Joint的作用是将两个刚体用固定的杠杆或者图钉连接起来，即当Pin Joint创建后，两锚点的距离始终保持不变。

Pin Joint的创建方式如下：

```
1  var pin = new cp.PinJoint(bodyA, bodyB, anchr1, anchr2);
```

其中bodyA和bodyB为要连接的两个物体，均为cp.Body类型。anchr1和anchr2分别为锚点相对于bodyA和bodyB重心的相对坐标，皆为cp.Vect类型。另外，我们可以通过pin.dist获取两锚点间的距离。

14.5.3　Slide Joint

Slide Joint类似于Pin Joint，不同的是它拥有最短和最长距离的限制，即两锚点的距离可以在最短和最长距离间滑动。

Slide Joint的创建方式如下：

```
1  var slide = new cp.SlideJoint(a, b, anchr1, anchr2, min, max);
```

其中min和max是锚点间的最短/最长距离限制，均为Number类型，其他参数同cp.PinJoint。

14.5.4　Pivot Joint

Pivot Joint也类似于Pin Joint，不同的是Pivot Joint使用单个锚点固定两个刚体，也可以理解为它是Pin Joint两锚点间的距离为零的特殊形式。

Pivot Joint有两种创建方式，具体如下：

```
1  // anchr为世界坐标
2  var pivot1 = new cp.PivotJoint(a, b, anchr);
3  // anchr1和anchr2分别为相对于a和b的重心的相对坐标
4  var pivot2 = new cp.PivotJoint(a, b, anchr1, anchr2);
```

关于Pivot Joint，有一个很好用的技巧，那就是可以使用此约束来达到鼠标拖动刚体的效果。通过给鼠标创建一个刚体（我们暂且把它称作鼠标刚体），将需要被拖动的刚体跟鼠标刚体通过Pivot Joint连接起来，我们可以在鼠标按下时创建该约束，弹起后释放它，就可以实现鼠标点击拖动刚体的效果，具体可参见本章实例。

14.5.5　Groove Joint

Groove Joint类似于Pivot Joint，不同的是它相对于body1并非使用的是锚点而是线段，body2上的锚点可以在body1的线段上自由移动。

Groove Joint的创建方式如下：

```
1   var groove = new cp.GrooveJoint(bodyA, bodyB, groove_a, groove_b, anchr2);
```

其中，groove_a和groove_b为相对于bodyA的线段的起点和终点，anchr2表示相对于b的锚点。

14.5.6 Damped Spring

Damped Spring类似于Slide Joint，不同的是两锚点间的连接受到的是弹力的约束，就像在两刚体的锚点间连接了一个弹簧。

Damped Spring的创建方式如下：

```
1   var spring = new cp.DampedSpring(a, b, anchr1, anchr2, restLength, stiffness,
    damping);
```

其中，restLength是静止时锚点间的长度，为Number类型。stiffness表示杨氏模量（Young's Modulus），同样为Number类型，具体请查阅杨氏模量的相关定义（见参考资源）。damping为弹簧的阻尼，即弹簧的柔软程度，也为Number类型，推荐数值为0.0到1.0之间的数。

14.5.7 Damped Rotary Spring

Damped Rotary Spring类似于Damped Spring，不同的是它约束的是弧度而非距离。

Damped Rotary Spring的创建方式如下：

```
1   var rotarySpring = new cp.DampedRotarySpring(a, b, restAngle, stiffness, damping);
```

其中，restAngle为静止时两刚体间的角度，其余参数可参考cp.DampedSpring。

14.5.8 Rotary Limit Joint

Rotary Limit Joint类似于Slide Joint，不同的是两刚体间受到的约束是它们间最大和最小的弧度而非距离。

Rotary Limit Joint的创建方式如下：

```
1   var rotaryLimitJoint = new cp.RotaryLimitJoint(a, b, min, max);
```

其中，min和max为两刚体间最大和最小的弧度值，均为Number类型。

14.5.9 Ratchet Joint

Ratchet Joint的作用方式类似于套筒扳手。

Ratchet Joint的创建方式如下：

```
1   var ratchet = new cp.RatchetJoint(a, b, phase, ratchet);
```

其中，phase是当决定棘轮角度时的初始弧度位移，ratchet是两次击打间的弧度。

14.5.10 Gear Joint

Gear Joint的作用方式类似于齿轮，它保持着一对刚体恒定的角速度比。创建方式如下：

```
1  var gear = new cp.GearJoint(a, b, phase, ratio);
```

其中，phase为两刚体初始弧度位移，ratio为a与b的角速度比，我们也可以通过以下接口来改变它的角速度比：

```
1  gear.setRatio(ratio);
```

14.5.11 Simple Motor

Simple Motor保持着两刚体间恒定的角速度，其创建方式如下：

```
1  var monitor = new cp.SimpleMotor(a, b, rate);
```

其中，rate为a和b刚体间的恒定角速度。

14.6 空间

上面介绍了Chipmunk的刚体、形状和约束，但是仅仅拥有这些还不够，我们还需要一个场景来模拟它们，而这个场景就是空间。空间可由如下方式创建：

```
1  var space = new cp.Space();
```

Chipmunk并没有主动执行物理模拟，我们必须手动调用它才能使物理模拟运行起来（space为cp.Space的实例，下同）：

```
1  space.step(dt); // dt为间隔时间戳，代表一次物理帧所经历的时间
```

通常，应该将此函数调用放在游戏帧回调当中，让游戏在每帧都执行一次物理帧。一般情形下，我们推荐使用一致的间隔时间戳来设置此值，这样能够使物理模拟更加稳定，但是如果游戏帧数过低，物理模拟速度也会变得很慢。如果我们的游戏跟时间关联较大，这种做法也是不能接受的。不过可以使用一些技巧，既可以使物理帧每帧的时间固定，又可以使物理时间跟游戏时间同步，相关代码如下所示：

```
1  var timeRemain = 0;
2  var stepTime = 1 / 60.0;
3  node.update = function(dt){
4      timeRemain += dt;
5      while(timeRemain - stepTime >= 0){
6          space.step(stepTime);
7          timeRemain -= stepTime;
8      }
9  }
```

cp.Space提供了几个全局设置的属性参数，它们将会影响到整个物理模拟的精度和表现，如下。

- **gravity**：物理空间重力，值为一个`cp.v(x, y)`类型的值。
- **iterations**：`Number`类型，默认值为10，表示迭代次数。此迭代次数将会影响到模拟的精度，更高的数值会使模拟变得更加精确、完美。刚体看上去更加坚硬，但是会消耗更多的CPU，较低的数值会使模拟变得更加模糊，刚体看上去也更加柔软，但是能节省CPU时间，所以最好根据自己的需要调整它。
- **idleSpeedThreshold**：`Number`类型，默认值为0，表示静止速度临界值， 即刚体被视为静止的速度临界值。 Chipmunk将根据重力设置自动设定此临界值。
- **sleepTimeThreshold**：`Number`类型，表示睡眠检测临界值，用来控制刚体静止后多长时间进入sleep状态（相关介绍详见14.3.2节）。其默认值为`Infinity`，代表关闭此算法，设置此值将隐式开启`space.enableContactGraph`。
- **collisionSlop**：`Number`类型，默认为0.1，表示碰撞刚体间的穿透鼓励值，如果你的物理模拟质量很差，可以在不会出现明显的碰撞重叠的前提下尽可能增加此值。
- **collisionBias**：`Number`类型，该参数决定了刚体重叠后分离的速度。Chipmunk会让快速移动的刚体重叠，然后再修复重叠。重叠基本上是不可避免的，此值代表了每秒重叠的错误保留百分比偏值，不推荐使用0，默认为`Math.pow(1, -0.1, 60)`，指的是在60帧的前提下，每秒修复10%的重叠。
- **collisionPersistence**：表示保存的碰撞帧数量，基本不需要修改。
- **enableContactGraph**：`Boolean`类型，表示是否在每个步骤重建碰撞图，默认为`false`，会略微提升一些性能。当`space.sleepTimeThreshold`开启时，将被隐式开启。注意，如果你要使用`body.eachArbiter`函数，必须开启此值。
- **damping**：阻尼，你可以用它来模拟空气阻力等。

此外，`cp.Space`还提供了`staticBody`变量，如14.3.2节介绍的那样，你可以直接使用此值而不需要自己去创建静态刚体。

下面我们来关注下`cp.Space`类对刚体、形状以及约束的管理接口。

添加接口的代码如下：

```
1  // 注意，你不能添加静态刚体到space当中
2  space.addBody(body);             // 添加[刚体]
3  space.addShape(shape);           // 添加[形状]
4  space.addStaticShape(shape);     // 添加[静态形状]
5  space.addConstraint(constraint);// 添加[约束]
```

移除接口的代码如下：

```
1  // 注意，移除刚体时，你必须先把刚体相关的形状和约束移除
2  space.removeBody(body);
3  space.removeShape(shape);
4  // 同样，你也可以使用space.removeShape来移除静态形状
5  space.removeStaticShape(shape);
6  space.removeConstraint(constraint);
```

查询接口的代码如下：

```
1  space.containsBody(body);
2  space.containsShape(shape);
3  space.containsConstraint(constraint);
```

遍历接口的代码如下，注意不要在遍历时做添加或删除操作：

```
1  space.eachBody(func);
2  space.eachShape(func);
3  space.eachConstraint(func);
```

cp.Space还拥有一些与碰撞检测或者查询相关的接口，碰撞检测相关的接口，如下所示（具体见14.7节）：

```
1  space.useSpatialHash(dim, count);
2  space.setDefaultCollisionHandler(begin,preSolve,postSolve,separate);
3  space.addCollisionHandler(a, b, begin,preSolve,postSolve,separate);
4  space.removeCollisionHandler(a, b);
5  space.lookupHandler(a, b);
6  space.addPostStepCallback(func);
```

查询相关接口如下（具体见14.8节）：

```
1  space.pointQuery (point, layers, group, func);
2  space.pointQueryFirst(point, layers, group);
3  space.nearestPointQuery(point, maxDistance, layers, group, func);
4  space.nearestPointQueryNearest(point, maxDistance, layers, group);
5  space.segmentQuery(start, end, layers, group, func);
6  space.segmentQueryFirst(start, end, layers, group);
7  space.bbQuery(bb, layers, group, func);
8  space.shapeQuery(shape, func);
```

14.7　碰撞检测

碰撞检测，顾名思义，就是用于检测刚体间是否发生碰撞，是物理引擎中非常重要的一环，同时它涉及的算法也非常复杂。所幸的是，作为接口的使用者，我们并不需要过多地去担心这些，只要知道如何使用它们就可以了。本节将对Chipmunk中的碰撞检测进行探讨。

14.7.1　空间索引

在空间中，如果通过for循环去遍历每一个刚体是否与其他刚体发生碰撞，运行时间往往是不可接受的，所以在碰撞检测的第一步，Chipmunk会使用高层次空间算法来快速找出哪些刚体彼此靠近，应该被检查碰撞。简单来说，就是在检查碰撞前先做一层筛选，能够大大地提升算法的速度。目前，Chipmunk支持两种空间索引（spatial index）方式：轴对齐包围盒和空间散列。在默认情形下，Chipmunk使用的是轴对齐包围盒作为空间索引方式，但是如果空间中有许多大小大致相同的形状，使用空间散列将能够提升碰撞检测的运算速度。下面我们就来看一下如何使用空间散列：

```
1  space.useSpatialHash(dim, count);
```

其中，参数dim为散列格大小（正方形），count为建议的最小散列格数目，dim的推荐设置值是所有形状大小的平均值，count的推荐设置值为10倍的形状数目，你可以从这个设置开始调优。注意，dim设置过小会导致形状插入到过多的散列格当中，而设置过大会导致一个散列格插入过多的形状，均会造成性能损失。而count设置过小会造成过多的误报，count设置过大会导致浪费过多的高速缓存和内存，所以你必须谨慎地设置这两个值。

下面我们就来看一下空间散列的示例，图14-4中的灰色正方形方块代表了散列格，散列格的颜色越深就代表了越多的物体被映射到散列格当中（示例来自 Chipmunk官网）。

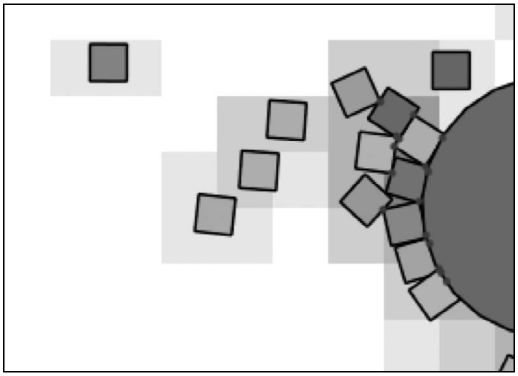


图14-4　空间散列示例

图14-5做了一个比较好的dim值估算，每个散列格的刚体个数较为均匀。

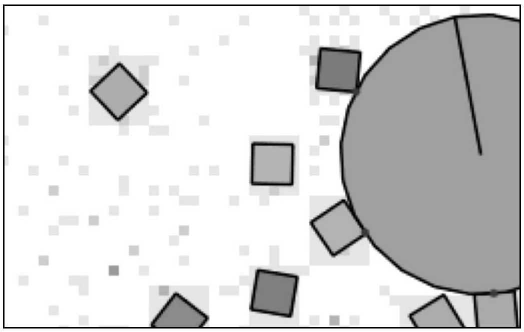


图14-5　dim值估算

图14-5使用了过小的dim值，导致单个刚体被插入到过多的散列格当中。而图14-6使用了过大的dim值，导致单个散列格插入过多的刚体，将造成过多不必要的碰撞检测。

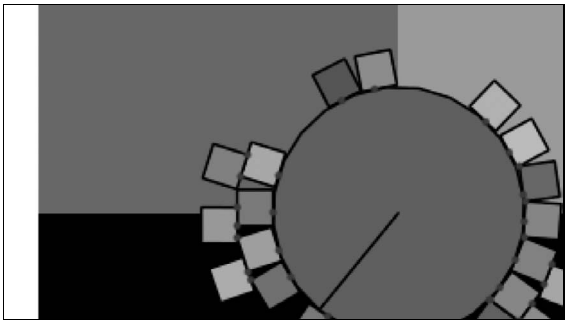


图14-6　使用过大的dim值

14.7.2　碰撞过滤

在空间索引检测出彼此靠近的形状对后，它们将会再执行一次额外的筛选。在此之前，Chipmunk会先做几个快速测试来检测形状是否会发生碰撞，这就是碰撞过滤（collision filter）。

Chipmunk将会做3种碰撞过滤测试。

- 包围盒测试（Bounding Box Test）：如果形状的包围盒没有重叠，那么形状就没有发生碰撞。通常，斜线线段的包围盒会造成很多误判，不过我们并不需要关心它。
- 组测试（Group Test）：形状处于相同的非零组时，将不会产生碰撞。
- 层测试（Layer Test）：只有当两形状的层的与运算（&）的结果不为零时，才会发生碰撞。

我们可以通过设置组和层来有选择地让不同的刚体发生碰撞或者穿透，下面将逐一学习它们。

- 组（Group）：形状处于相同的非零组时将不会产生碰撞，优先级高于层，即当形状处于不同组别时层碰撞过滤才会生效。默认值为`cp.NO_GROUP`，即为零。组可以用来忽略复杂物体不同部分的碰撞，比如把手臂安装到身体上，我们想让它们重叠而且不发生碰撞，我们就可以把它们设置在同一个组。
- 层（Layer）：只有当两形状的层的与运算（&）的结果不为零时，才会发生碰撞，通过此值你可以把形状划分成不同的类别，默认值为`cp.ALL_LAYERS`，它的所有二进制位均为1，即和所有其他层的物体都会发生碰撞。例如，我们有5类物体——玩家、敌人、玩家子弹、敌人子弹、墙。我们的需求为：玩家和玩家的子弹不发生碰撞，敌人和敌人的子弹不发生碰撞，子弹间均不发生碰撞，除敌人外其他同种物体均不发生碰撞，墙与任何物体均发生碰撞（示例来自Chipmunk官网）。

首先，我们先将除敌人外的同类物体均设置在同一非零组，即让除敌人外的所有同种物体间均不发生碰撞，如表14-2所示。

表14-2 为物体设置组

物体	玩家	敌人	玩家子弹	敌人子弹	墙
组	1	0	3	4	5

然后将它们分别部署在不同的层，它们与的结果均为零，即让不同物体间均不发生碰撞（二进制表示），如表14-3所示。

表14-3 为物体设置层

物体	玩家	敌人	玩家子弹	敌人子弹	墙
层	00001	00010	01000	01000	10000

然后根据我们的需求做调整，一步步开启不同物体间的碰撞规则（二进制表示），如表14-4所示。

表14-4 调整后的层

物体	玩家	敌人	玩家子弹	敌人子弹	墙
层	00001	00010	00110	01001	11111

最终结果为（十进制表示）如表14-5所示。

表14-5 最终结果（十进制）

物体	玩家	敌人	玩家子弹	敌人子弹	墙
组	1	0	3	4	5
层	1	2	6	9	31

此时即可达到我们想要的效果。

14.7.3 碰撞回调

前面我们学习了空间索引和碰撞过滤，然而这些并不足以让你完成一个有趣的物理游戏，很多时候你可能需要在刚体碰撞时做一些计算或者产生一些效果，比如子弹打到敌人时需要扣除敌人血量，炸弹碰到地面时需要产生爆炸等，这时就需要碰撞回调（collision callback）来完成这些功能。

Chipmunk提供了一组回调函数来处理碰撞事件，它们代表了不同的碰撞阶段。要让你的物理空间能监听物体碰撞，并在碰撞的时候实现对应的逻辑处理，需要两个步骤。首先，你需要给当前的空间添加碰撞回调，代码如下：

```
1  // 加载[物理空间]
2  loadPhysicsSpace : function(){
3      // ......
4      this.space.addCollisionHandler(121, 212,
```

```
5           this.collisionBegin.bind(this),
6           this.collisionPre.bind(this),
7           this.collisionPost.bind(this),
8           this.collisionSeparate.bind(this)
9       );
10 }
```

其中，121和212表示要监听的碰撞类型（这将在后面进一步介绍），其值可以为Number或者String类型。接着还需要实现碰撞回调的相关函数，代码如下：

```
1  // 刚开始接触时的回调
2  collisionBegin : function(arbiter, space){
3      return true;
4  },
5  // 接触的每次step的回调
6  collisionPre : function(arbiter, space){
7      return true;
8  },
9  // 接触并且碰撞响应已经处理的回调
10 collisionPost : function(arbiter, space){
11 },
12 // 开始分离的step的回调
13 collisionSeparate : function(arbiter, space){
14 }
```

下面简要介绍一下上述代码中涉及的函数。

- **collisionBegin(arbiter, space)**：两形状刚开始接触时的回调，每次接触只触发该回调一次，返回true会使形状正常产生碰撞。如果返回false，会使形状忽略碰撞而互相穿透，并且preSolve和postSolve回调均不会被触发。
- **collisionPre(arbiter, space)**：两形状接触的每次step产生回调，你可以选择返回false来忽略本次的碰撞，或者返回true使Chipmunk正常处理碰撞。并且，你还可以在此回调函数当中获取出对应的Shape，然后调用cp.Shape的setFriction(friction)和setElasticity(elasticity)等函数来重新定义摩擦系数和弹性系数等，具体请参照14.5节。示例代码如下：

```
1  // 接触的每次step的回调
2  collisionPre : function(arbiter, space){
3      var shapeB = arbiter.b;
4      shapeB.setElasticity(1.0);
5
6      var bodyB = arbiter.body_b;
7      bodyB.applyImpulse(cp.v(0, 2000), cp.v(0, 0));
8      return true;
9  }
```

- **collisionPost(arbiter, space)**：两形状接触并且碰撞响应已经被处理，也是在两形状接触的每次step产生回调。注意，sensor类型的形状不会触发此回调，因为它们从来不会进行碰撞处理。Chipmunk提供此回调，可以方便我们在每次碰撞响应处理后计算

冲力和动能，方便我们游戏计算碰撞声音的大小或者碰撞产生的损伤程度，具体请参照14.5节。

- **collisionSeparate(arbiter, space)**：两形状开始分离的step产生一次回调。需要注意的是，它和begin回调是一个回调对，即产生一次begin回调，必然会产生一次separate回调，所以形状在接触时被移除时也会产生一次该回调。

这4个回调函数的参数均为arbiter和space，arbiter为cp.Arbiter的对象，即碰撞对（这将在14.7.4节中介绍），space为cp.Space，即碰撞发生的空间。

注意，这4个回调通常在space.step函数运行时发生，所以你不能在这些回调中调用add/remove函数。如果此时有add/remove的需求，应当使用space.addPostStepCallback函数来处理。例如，现在有一个需求是这样的：当刚体的Y坐标小于屏幕Y坐标的一半时，移除掉当前刚体以及形状，实现代码应该如下：

```
1  collisionPre : function(arbiter, space){
2      var shapeB = arbiter.getB();
3      var bodyB = arbiter.body_b;
4      if (bodyB.getPos().y <= cc.winSize.height / 2) {
5          this.space.addPostStepCallback(function(){
6              this.space.removeBody(bodyB);
7              this.space.removeShape(shapeB);
8          }.bind(this));
9      }
10     return true;
11 }
```

addPostStepCallback允许我们在当前物理帧结束后执行一次回调函数，注意该函数每次注册都只会执行一次。此接口的目的主要是让你能够在物理帧结束时安全地删除对象。

接下来，我们就来看看在cp.Space中与注册和使用这些回调相关的接口。

设置默认的碰撞回调函数，这些函数将会接收空间内所有碰撞所产生的回调，代码如下：

```
1  space.setDefaultCollisionHandler(begin, preSolve, postSolve, separate);
```

添加两个类型的碰撞回调函数，这些函数只会接收碰撞类型a和碰撞类型b碰撞时产生的回调，a和b可为Number类型，也可以为String类型。接口如下：

```
1  space.addCollisionHandler(a, b, begin, preSolve, postSolve, separate);
```

如果设定a的值为10086、b的值为10010，并且物理空间（space）中有个ShapeA和ShapeB，它们的碰撞类型分别10086和10010，那么，当ShapeA和ShapeB产生碰撞时，便会触发preSolve、postSolve和separate回调函数。shape的碰撞类型可以通过如下方式设定，其中type的值对应a或b：

```
1  shape.setCollisionType(type);
```

移除两形状的碰撞回调，代码如下：

```
1  space.removeCollisionHandler(a, b);
```

查找两形状的碰撞回调句柄，如果找不到，则会返回默认的回调函数句柄。返回的变量类型为CollisionHandler，它拥有begin、preSolve、postSolve和separate这4个变量，它们分别保存了对应的4个回调函数，代码如下：

```
1  space.lookupHandler(a, b);
```

14.7.4　碰撞对

在前面的章节中，我们多次接触到cp.Arbiter类，下面我们就来学习一下它。

cp.Arbiter是Chipmunk用于处理碰撞检测的一个重要的类，Chipmunk不允许我们创建该类，但是从前面的章节中可以看到，我们有许多种方式来获取它。它包含了许多重要的碰撞检测相关信息，下面我们来看看它都拥有哪些接口（arbiter为cp.Arbiter类的对象，下同）：

```
1  // 获取形状a
2  arbiter.getA();
3  // 获取形状b
4  arbiter.getB();
5  // 返回Array类型，包含了形状a和形状b
6  arbiter.getShapes();
```

下面的接口均为计算并返回对应结果的接口，只有在postSolve回调、postStepCallback回调和body.eachArbiter的回调函数中使用才能得到正确的结果：

```
1  // 计算碰撞对的总冲力
2  arbiter.totalImpulse();
3  // 计算包括摩擦力在内的碰撞对的总冲力
4  arbiter.totalImpulseWithFriction();
5  // 计算总的能量丢失的数值
6  arbiter.totalKE();
```

14.8　查询

有时候我们需要知道在某个点、某条路径或者某个区域内是否有刚体的存在，这时就需要用到查询（query）。注意，每个查询都会让你传入想要查询的组和层，如果不想做过滤，可是把组设置为cp.NO_GROUP，层设置为cp.ALL_LAYERS。

14.8.1　点查询

点查询的相关接口如下：

```
1  // 点查询，并在该点的所有形状逐一产生一次回调func(shape)
2  space.pointQuery (point, layers, group, func);
3  // 点查询，但只返回查询到的第一个形状，查询不到时返回null
4  space.pointQueryFirst(point, layers, group);
5  // 最近点查询，也可以称作圆查询，它以point为圆心，以maxDistance为半径查询所有的形状并逐一产
   // 生一次回调func(shape)
```

```
6  space.nearestPointQuery(point, maxDistance, layers, group, func);
7  // 最近点查询，但是只返回查询到的第一个形状的NearestPointQueryInfo结构
8  space.nearestPointQueryNearest(point, maxDistance, layers, group);
```

此外，我们还可以通过调用cp.Shape的查询函数对单个形状进行点查询：

```
1  // 点查询，当形状在该点上时返回NearestPointQueryInfo结构，不在时返回null
2  shape.pointQuery(p);
3  // 点查询，返回NearestPointQueryInfo结构
4  shape.nearestPointQuery(p);
```

NearestPointQueryInfo结构的属性如下：

```
1  info.shape; // 形状
2  info.p;     // 形状上距离查询点最近的点
3  info.d;     // 到查询点的距离，当查询点在形状内部时，该值为负
```

14.8.2 线段查询

线段查询的相关API如下：

```
1  // 线段查询，start和end为查询的线段的起点和终点，查询到的所有形状逐一产生一次回调func(shape)
2  space.segmentQuery(start, end, layers, group, func);
3  // 线段查询，但是只返回查询到的第一个形状的SegmentQueryInfo结构
   space.segmentQueryFirst(start, end, layers, group);
```

我们还可以通过调用cp.Shape的查询函数对单个形状进行线段查询：

```
1  shape.segmentQuery(a, b);
```

其中，a和b为查询线段的起点和终点，函数将返回SegmentQueryInfo结构，当形状与线段不相交时返回null。

SegmentQueryInfo结构体的源码如下：

```
1  var SegmentQueryInfo = function(shape, t, n){
2      this.shape = shape;
3      this.t = t;
4      this.n = n;
5  };
```

其中t表示线段查询的归一化距离，在[0, 1]范围内，即被命中位置在查询线段起点和终点的百分比位置，如果线段查询起点在形状内部，此值为0。n为命中点的法向量，如果线段查询起点在形状内部，此值为cp.vzero。

14.8.3 区域查询

区域查询的相关API如下：

```
1  // AABB查询，bb为cp.BB类型，查询到的所有形状逐一产生一次回调
2  space.bbQuery(bb, layers, group, func);
3  // 形状重叠查询，shape为cp.Shape类型，在查询到的所有形状逐一产生一次回调
4  space.shapeQuery(shape, func);
```

14.9　实例——拖动刚体的实现

拖动指的是抓住物体的某一点，以这一点作为托动力的作用点，物体会在托动力的作用下产生位移、旋转等，这和Cocos2d-JS中通过触摸事件移动普通精灵有所不同。下面我编写一个实例（本实例见随书源码src/unit14_chipmunk），在这个实例中有一个刚体，你可以拖动此刚体，但是不能让此刚体碰到物理空间的边缘，如若碰到，则会删除此刚体、刚体的形状以及与刚体绑定在一起的精灵。

首先，我定义了一个Layer，在这个Layer下新建一个物理空间，代码如下：

```
1 // 加载[物理空间]
2 loadPhysicsSpace : function(){
3     this.space = new cp.Space();            // 创建[物理空间]
4     this.space.gravity = cp.v(0, -150);     // 设置[物理空间重力]
5     this.space.sleepTimeThreshold = 0.5;    // 设置[睡眠检测临界值]
6
7     var thickness = 100;    // 物理空间围墙厚度
8     var staticBody = this.space.staticBody; // 创建[静态刚体]
9     var walls = [
10        new cp.SegmentShape(staticBody, cp.v(-thickness, -thickness), cp.v
          (cc.winSize.width, -thickness), thickness),  // 底部
11        new cp.SegmentShape(staticBody, cp.v(-thickness, -thickness), cp.v
          (-thickness, cc.winSize.height), thickness), // 左边
12        new cp.SegmentShape(staticBody, cp.v(cc.winSize.width + thickness,
          -thickness), cp.v(cc.winSize.width + thickness, cc.winSize.height),
          thickness ),      // 右边
13        new cp.SegmentShape(staticBody, cp.v(-thickness, cc.winSize.height +
          thickness), cp.v(cc.winSize.width, cc.winSize.height + thickness),
          thickness)       // 顶部
14    ];
15
16    // 创建[围墙]
17    for(var i = 0; i < walls.length; i++){
18        var shape = walls[i];
19        shape.setElasticity(0.5);   // 设置[围墙的弹性系数]
20        shape.setFriction(0);       // 设置[围墙的摩擦力]
21        shape.setCollisionType(this.collisionType.walls);
22        this.space.addStaticShape(shape); // 添加[围墙到物理空间中]
23    }
24
25    // 添加[碰撞监听]
26    this.space.addCollisionHandler(
27        this.collisionType.walls,
28        this.collisionType.tapBody,
29        this.collisionBegin.bind(this),
30        this.collisionPre.bind(this),
31        this.collisionPost.bind(this),
32        this.collisionSeparate.bind(this)
33    );
34 }
```

在第7行代码中，我定义了一个thickness变量，用来表示物理空间围墙的厚度。实际上，它的值为0也是可以的，但是如若刚体的运动速度过快，可能会导致刚体穿过围墙，这显然是你所不愿意看到的效果，那么将围墙的厚度设置宽一点是一种解决方案。

在第21行代码中，我设置了围墙的碰撞类型为this.collisionType.walls。实际上，this.collisionType是一个空对象，它用来模拟枚举。this.collisionType.walls的值为String类型，除了walls之外，我还为this.collisionType对象定义了tapBody成员，用来表示本实例中要拖动刚体的形状的碰撞类型，代码如下：

```
1  // 加载[配置]
2  loadConfig : function(){
3      this.collisionType = {};
4      this.collisionType.walls = "walls";
5      this.collisionType.tapBody = "tapBody";
6  }
```

最后，在第25行到第33行代码中，我给碰撞类型为this.collisionType.walls和this.collisionType.tapBody的形状添加了碰撞监听，这样当这两个形状发生碰撞时，我们便可在碰撞的回调函数中移除掉对应的刚体、形状以及和刚体绑定的精灵。

接下来，我封装一个可以拖曳的物理精灵类，它继承自cc.PhysicsSprite，我将它命名为MousePhysicsSprite。在MousePhysicsSprite类中，我为它定义了3个成员属性，并且注册了触摸事件，相关代码如下：

```
1  var MousePhysicsSprite = cc.PhysicsSprite.extend({
2      space : null,           // 物理空间引用
3      mouseBody : null,       // 鼠标刚体
4      pivotJoint : null,      // pivot关节
5      ctor : function(fileName, rect){
6          this._super(fileName, rect);
7          // 注册[事件]
8          this.registerEvent();
9      },
10     onEnter : function(){
11         this._super();
12         this.space = this.getBody().space;
13     },
14     // 注册[事件]
15     registerEvent : function(){
16         this.cardTouchListener = cc.EventListener.create({
17             event            : cc.EventListener.TOUCH_ONE_BY_ONE,
18             swallowTouches   : true,
19             onTouchBegan     : this.onTouchBegan,
20             onTouchMoved     : this.onTouchMoved,
21             onTouchEnded     : this.onTouchEnded,
22             onTouchCancelled : this.onTouchCancelled
23         });
24         cc.eventManager.addListener(this.cardTouchListener, this);
25     }
26 });
```

在上述代码中，成员属性space（第2行）指向物理空间，mouseBody（第3行）表示鼠标刚体。需要注意的是，Chipmunk中只有刚体和静态刚体这两种刚体，mouseBody实际上是一个静态刚体，只是我将它称为鼠标刚体。另外的pivotJoint为Pivot类型的关节约束。

第8行代码给MousePhysicsSprite注册了触摸事件，第22行代码为触摸事件的onTouchCancelled回调，此回调一般在触摸事件失去焦点的时候被触发到，例如设备接到电话打入等，如若不实现onTouchCancelled回调，则此示例会造成bug产生。

接下来，我们来看看触摸事件相关回调函数的实现，这也是本示例的核心。onTouchBegan的实现代码如下：

```
1  onTouchBegan: function (touch, event) {
2      var target = event.getCurrentTarget();      // 获取[当前事件源]
3      var position = touch.getLocation();         // 获取[触摸点位置]
4      var locationInNode = target.convertToNodeSpace(position);
5      var size = target.getContentSize();
6      var rect = cc.rect(0, 0, size.width, size.height);
7      if (!(cc.rectContainsPoint(rect, locationInNode))) {
8          return false;
9      }
10
11     var mouseBody = new cp.StaticBody();        // 创建[静态刚体]
12     mouseBody.setPos(cp.v(position.x, position.y));// 设置[刚体坐标]
13     var joint = new cp.PivotJoint(target.getBody(), mouseBody, cp.v(position.x,
       position.y));
14     target.space.addConstraint(joint);          // 添加[约束]
15     target.mouseBody = mouseBody;
16     target.pivotJoint = joint;
17
18     return true;
19 }
```

第2行到第9行代码用于判断触摸点是否触摸到当前的精灵，若没触摸到，则返回。

第11行代码创建了一个静态刚体，也就是前面我所说的鼠标刚体。通过改变它的坐标，再为它和要拖动的刚体加上Pivot关节约束，便可实现鼠标拖动刚体的效果。在第13行代码中，我创建出了Pivot类型的关节约束，这个约束作用在MousePhysicsSprite的刚体（target.getBody()）和鼠标刚体上（mouseBody），并在第14行代码中将此约束添加到物理空间space中去。而第15行和第16行代码则让MousePhysicsSprite类的成员属性mouseBody和pivotJoint分别指向对应的刚体和约束，这两个成员属性在后面的onTouchMove等函数中也会用到。下面我们来看看onTouchMoved函数的实现，具体如下：

```
1  onTouchMoved : function (touch, event) {
2      var target = event.getCurrentTarget();
3      if(target.mouseBody) {
4          var position = touch.getLocation();
5          target.mouseBody.setPos(cp.v(position.x, position.y));
6      }
   }
```

可以看到，onTouchMoved函数内部的实现比较简单，实际上，它只做了一件事情，就是让mouseBody刚体的坐标始终为触摸点的坐标。因为此时space中有了pivotJoint约束，pivotJoint约束又作用在鼠标刚体和当前精灵所绑定的刚体上，所以当改变鼠标刚体的坐标时，当前精灵所绑定的刚体便受到了pivotJoint约束的作用，从而产生了拖动的效果。

最后，在onTouchEnded函数内部，应该移除掉pivotJoint约束。而onTouchCancelled函数则直接调用onTouchEnded函数即可，代码如下：

```
1  onTouchEnded : function (touch, event) {
2      var target = event.getCurrentTarget();
3      if(target.mouseBody){
4          target.space.removeConstraint(target.pivotJoint);
5      }
6  },
7  onTouchCancelled : function (touch, event) {
8      var target = event.getCurrentTarget();
9      target.onTouchEnded(touch, event);
10 }
```

至此，MousePhysicsSprite编写完毕。接下来，便可以在Layer中创建出可拖动的刚体精灵了，代码如下：

```
1  // 加载[可以拖曳的刚体]
2  loadTapBody : function(){
3      var node = new MousePhysicsSprite(res.sh_node_128_png);
4      this.addChild(node);
5      this.mousePhysicsSprite = node;
6
7      var mess = 10;      // 质量
8      var body = new cp.Body(mess, cp.momentForBox(mess, node.width, node.height));
9      this.space.addBody(body);
10     node.setBody(body);          // 精灵设置刚体
11     body.setPos(cp.v(cc.winSize.width / 2, cc.winSize.height / 2));
12     body.sprite = node;          // 给body添加sprite属性，指向node
13
14     var shape = new cp.BoxShape(body, node.width, node.height);
15     this.space.addShape(shape); // 物理空间添加形状
16     shape.setElasticity(0.5);   // 设置[弹性系数]
17     shape.setFriction(0.5);     // 设置[摩擦力]
18     shape.setCollisionType(this.collisionType.tapBody);
19     return body;
20 }
```

上面的代码可分为3个部分，分别为创建刚体精灵（第3行到第5行）、创建刚体（第7行到第12行）和创建形状（第14行到第18行）。在第3行代码中，我创建出了之前编写的MousePhysics-Sprite。特别值得注意的是第12行代码，我给刚体添加了sprite属性，让它指向MousePhysicsSprite的实例对象node。这样做的目的是为了在后面的碰撞回调函数中获取出碰撞刚体时，可以判断刚体是否有sprite属性，若有，则做删除操作。碰撞开始接触的逻辑处理代码如下：

```
1  // 刚开始接触时的回调
2  collisionBegin : function(arbiter, space){
3      var bodyArray = [];
4      bodyArray.push(arbiter.body_a);
5      bodyArray.push(arbiter.body_b);
6
7      var body = null;
8      for (var i = 0; i < bodyArray.length; i++) {
9          body = bodyArray[i];
10         if (body.sprite === this.mousePhysicsSprite) {
11             var callFunc = function(theBody){
12                 // 移除[Shape]
13                 theBody.eachShape(function(shape){
14                     this.space.removeShape(shape);
15                 }.bind(this));
16                 // 移除[Body]
17                 this.space.removeBody(theBody);
18                 // 移除[Sprite]
19                 this.removeChild(theBody.sprite);
20             }.bind(this, body);
21             // 回调函数
22             this.space.addPostStepCallback(callFunc);
23         }
24     }
25     return true;
26 }
```

在上述代码中，第3行到第5行代码构建了一个刚体数组来保存碰撞的两个刚体。然后对这个数组进行遍历，这样做是因为刚体在碰撞的时候，我们并不知道是body_a和body_b分别指向哪个刚体，即无法知道到底是body_a撞body_b，还是body_b撞body_a。为了安全起见，应当都对它们做一个遍历。

在loadTapBody函数中，我给MousePhysicsSprite所要绑定的刚体添加了一个sprite属性，这个属性便在上面的第10行代码中起到了作用。body.sprite等于this.mousePhysics-Sprite，那么此刚体便是loadTapBody函数中所创建的那个刚体，我们就可以删除掉刚体精灵、刚体以及形状了。前面说过，collisionBegin等4个回调是在space.step函数运行时发生的，所以不能在这些回调中调用add/remove函数，应当使用space.addPostStepCallback函数来处理。space.addPostStepCallback函数接受一个回调函数，那么在第11行到第20行代码中，我定义了此回调函数。值得注意的是，在第20行代码中，我把body传递给callFunc函数。另外值得说明的就剩下第13行代码了，你应该知道了，每个刚体上可以添加多个形状，所以应当将它们一一遍历出来删除掉。

至此，本章实例结束，你可以运行随书代码，运行结果如图14-7所示，其中，在包围盒内部的小点为pivotJoint，而外部的小点为鼠标刚体（静态刚体）。

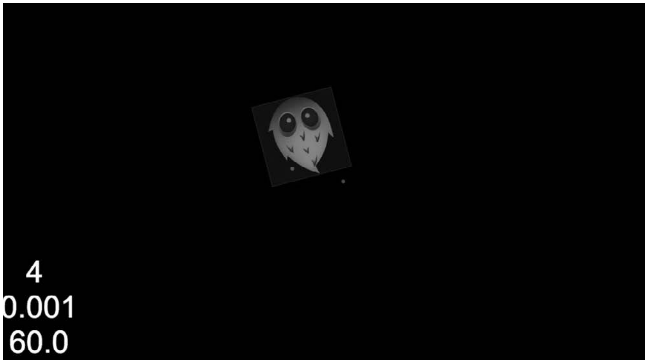


图14-7　实例运行结果

14.10　小结

通过本章的学习，我们知道了在Cocos2d-JS中支持了两个2D的物理引擎，分别是Chipmunk和Box2D。因为在JSB上，Box2D还没有被支持，所以在开发中，我们一般使用Chipmunk物理引擎。Chipmunk物理引擎中有刚体的概念，刚体分为静态刚体和动态刚体。此外，在刚体上可以添加形状等，形状分为圆形、线段、凸多边形，形状上还可以设置摩擦系数和弹性系数等。除了刚体和形状外，Chipmunk中提供了各种各样的约束，我们可以通过这些约束模拟出很多效果，例如绳子和齿轮等。最后，在本章实例中，我们一起完成了拖曳刚体的演示。

14.11　参考资源

本章的参考资源如下所示。

- 转动惯量：https://zh.wikipedia.org/wiki/转动惯量。
- 杨氏模量：https://zh.wikipedia.org/wiki/杨氏模量。
- 库仑摩擦力：http://baike.baidu.com/view/4309065.htm。
- 动能定理：https://zh.wikipedia.org/wiki/动能定理。

第 15 章 网络编程

在前面的章节中，你已经基本掌握了Cocos2d-JS的各项技能，单纯地运用那些技术也足够开发出一款质量不错的单机游戏。但是，随着手游行业如火如荼的发展，大部分单机类的游戏早已满足不了玩家的内心需求。网络游戏相对于单机游戏而言，可以给玩家带来更多的互动以及乐趣，维基百科对网络游戏的定义是这样的：

网络游戏（英语：Online Game），也称“在线游戏”，一般指多名玩家通过计算机互联网进行交互娱乐的电子游戏，以游戏运行软件分类，一般包含客户端下载的大型多人在线游戏，网页即开即玩的网页游戏以及社交网站游戏等。部分游戏能通过连接网络服务器进行联机对战、在线云存档。网络游戏有战略游戏、动作游戏、体育游戏、格斗游戏、音乐游戏、竞速游戏、网页游戏和角色扮演游戏等多种类型，不过也有少数在线网络单人游戏。

本章将围绕Cocos2d-JS的网络编程展开，但是网络编程所涉及的知识体系非常庞大。因为篇幅有限，所以本章的知识点也是有限的，你可以将其作为一块敲门砖。至于更详细的网络编程方面的知识，我建议你去购买专业的网络编程图书或者上网查阅相关的资料。

本章内容：

- 网络编程相关概念
- TCP/IP参考模型
- 基于HTTP协议的通信
- 基于WebSocket的通信
- 基于SocketIO的通信
- 实例——使用Node.js+Socket.IO实现游戏聊天系统

15.1 网络编程相关概念

在正式学习网络编程之前，首先要了解一下网络编程的一些基础概念。

OSI 参考模型

提到网络，就不得不提一下OSI参考模型。在网络开放之初，许多大型公司都有自己的网络技术。那时的网络没有一个统一的规范，每个公司都自己制定自己的网络通信规则，同一公司内部的计算机可以相互连接，但是不同公司的计算机却无法连接。为了更好地普及网络，并使网络上的机器都能相互连接，国际标准化组织（ISO）提出了这样一个模型：OSI参考模型。

OSI参考模型，又叫做开放式系统互联通信参考模型（Open System Interconnection Reference Model，ISO/IEC 7498-1），它是一种概念模型，由国际标准化组织（ISO）1974年提出，并在1983年正式启用，是一个试图使各种计算机在世界范围内互连为网络的标准框架。

OSI将网络划分成7层，从下往上分别是物理层、数据链路层、网络层、传输层、会话层、表示层以及应用层，每个层都有自己的协议和功能，使得数据可以在不同的系统、网络之间进行可靠传输，OSI的数据封装如图15-1所示。

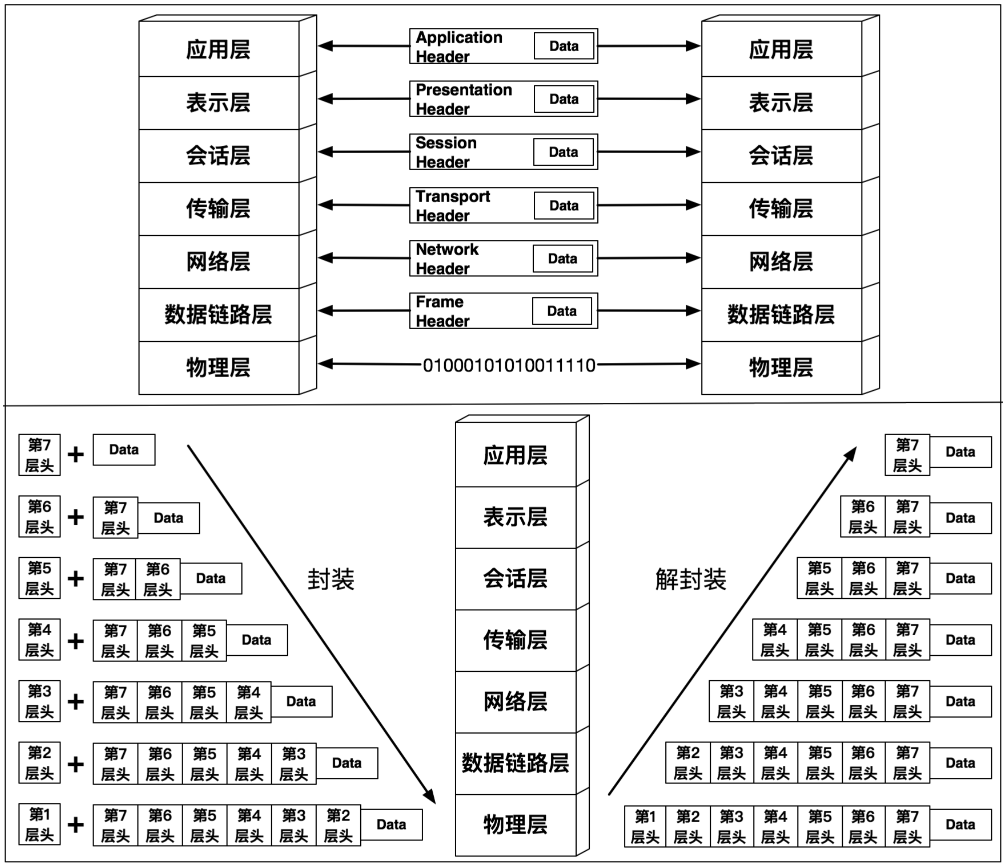


图15-1　OSI的数据封装

15

在OSI网络模型中，数据的传输方式是，由发送端从上往下传输，在传输的同时每一层将会对数据包进行处理封装，附带上每一层的数据头信息，然后再由物理层发送到接收端的物理层，再由接收端的物理层从下往上传递，层层解包，最后由接收端的应用层接收。所以，不难看出，发送端和接收端中相同的层需要处理的数据是一样的，你可以把发送端和接收端相同的层之间视为拥有了一个虚拟的通路，便于你理解和编程。

接下来，从下往上简要看一下各个层的功能。

1. 物理层

OSI模型的最低层或第一层，该层包括物理连网媒介，如电缆连线连接器。物理层（physical layer）的协议产生并检测电压以便发送和接收携带数据的信号。在你的桌面PC 上插入网络接口卡，你就建立了计算机连网的基础。换言之，你提供了一个物理层，尽管物理层不提供纠错服务，但它能够设定数据传输速率并监测数据出错率。网络物理问题，如电线断开，将影响物理层。用户要传递信息，就要利用一些物理媒体，如双绞线、同轴电缆等，但具体的物理媒体并不在OSI的7层之内，有人把物理媒体当做第0层。物理层的任务就是为它的上一层提供一个物理连接，以及它们的机械、电气、功能和过程特性，如规定使用电缆和接头的类型、传送信号的电压等。在这一层，数据还没有被组织，仅作为原始的位流或电气电压处理，单位是比特。工作在这一层的物理设备有网卡、网线、集线器、中继器、调制解调器以及各种无线、蓝牙技术等。

2. 数据链路层

OSI模型的第二层——数据链路层（datalink layer）控制着网络层与物理层之间的通信。它的主要功能是如何在不可靠的物理线路上进行可靠的数据传递。为了保证传输，从网络层接收到的数据会被分割成特定的可被物理层传输的帧。帧是用来移动数据的结构包，它不仅包括原始数据，还包括发送方和接收方的物理地址以及纠错和控制信息。其中的地址确定了帧将发送到何处，而纠错和控制信息则确保帧无差错到达。如果在传送数据时，接收点检测到所传数据中有差错，就要通知发送方重发这一帧。

数据链路层的功能独立于网络及其节点和所采用的物理层类型，它也不关心是否正在运行Word、Excel或使用Internet。它在物理层提供比特流服务的基础上，建立相邻节点之间的数据链路，通过差错控制提供数据帧（frame）在信道上无差错的传输，并进行各电路上的动作系列。

数据链路层在不可靠的物理介质上提供可靠的传输，该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。

数据链路层协议的代表包括：SDLC、HDLC、PPP、STP、帧中继等。工作在这一层的设备有网桥、交换机等。

3. 网络层

OSI模型的第三层是网络层（network layer），其主要功能是将网络地址翻译成对应的物理地址，并决定如何将数据从发送方路由到接收方。

网络层通过综合考虑发送优先权、网络拥塞程度、服务质量以及可选路由的花费来决定从一个网络中节点A到另一个网络中节点B的最佳路径。由于网络层处理路由，而路由器因为连接网络各段，并智能指导数据传送，所以它属于网络层。

在网络中，“路由”是基于编址方案、使用模式以及可达性来指引数据的发送。网络层负责在源机器和目标机器之间建立它们所使用的路由。这一层本身没有任何错误检测和修正机制，因此，网络层必须依赖于端端之间的由DLL提供的可靠传输服务。

网络层用于本地LAN网段之上的计算机系统建立通信，它之所以可以这样做，是因为它有自己的路由地址结构，这种结构与第二层机器地址是分开的、独立的。这种协议称为路由或可路由协议。路由协议包括IP、Novell公司的IPX以及AppleTalk协议。

网络层是可选的，它只用于当两个计算机系统处于不同的由路由器分割开的网段这种情况，或者当通信应用要求某种网络层或传输层提供的服务、特性或者能力时。例如，当两台主机处于同一个LAN网段的直接相连这种情况，它们之间的通信只使用LAN的通信机制就可以了（即OSI参考模型的一二层）。

4. 传输层

传输层（transport layer）是OSI模型中最重要的一层，传输协议同时进行流量控制或是基于接收方可接收数据的快慢程度规定适当的发送速率。除此之外，传输层按照网络能处理的最大尺寸将较长的数据包进行强制分割。例如，以太网无法接收大于1500字节的数据包。发送方节点的传输层将数据分割成较小的数据片，同时对每一个数据片安排一序列号，以便数据到达接收方节点的传输层时，能以正确的顺序重组，该过程被称为排序。工作在传输层的一种服务是 TCP/IP协议套中的TCP（传输控制协议），另一项传输层服务是IPX/SPX 协议集的SPX（序列包交换）。

5. 会话层

会话层（session layer）负责在网络中的两节点之间建立、维持和终止通信。会话层的功能包括：建立通信链接，保持会话过程中通信链接的畅通，同步两个节点之间的对话，决定通信是否被中断以及通信中断时决定从何处重新发送。你可能常常听到有人把会话层称作网络通信的“交通警察”。当通过拨号向你的ISP（因特网服务提供商）请求连接到因特网时，ISP服务器上的会话层向你与你的PC客户机上的会话层进行协商连接。若你的电话线偶然从墙上插孔脱落时，终端机上的会话层将检测到连接中断并重新发起连接。会话层通过决定节点通信的优先级和通信时间的长短来设置通信期限。

6. 表示层

表示层（presentation layer）为应用程序和网络之间的翻译官，这里数据将按照网络能理解的方案进行格式化，这种格式化也因所使用网络的类型不同而不同。

表示层管理数据的解密与加密，如系统口令的处理。例如，在Internet上查询银行账户，使用的即是一种安全连接。你的账户数据在发送前被加密，在网络的另一端，表示层将对接收到的数

据解密。除此之外，表示层协议还对图片、视频、文本等文件格式信息进行解码和编码，例如MPEG和JPEG。

7. 应用层

应用层（application layer）负责对软件提供接口以使程序能使用网络服务。术语“应用层”并不是指运行在网络上的某个特别应用程序，应用层提供的服务包括文件传输、文件管理以及电子邮件的信息处理。应用层的协议有很多，常用的有SMTP、FTP、HTTP等，其中HTTP是手游编程中用得最多的协议。

15.2 TCP/IP 参考模型

OSI参考模型是一个理想的模型，然而它过于庞大、复杂，不利于工业化普及，所以长期被人所诟病。故此，大多的机器使用的却是另一种模型：TCP/IP参考模型。

TCP/IP参考模型不同于OSI参考模型，它是先有了协议，才制定的模型，它的概念划分远不比OSI模型，许多的功能描述和实现细节混合在一起，但是它的普及度和实用性却远比OSI模型高。

TCP/IP参考模型把网络划分成了4层，从下往上分别是网络接口层（link layer）、网络互连层（internet layer）、传输层（transport layer）和应用层（application layer）。它的应用层对应了OSI参考模型的应用层，传输层对应了OSI参考模型的传输层，网络互连层对应了OSI参考模型的网络层，网络接口层对应了OSI参考模型的数据链路层和物理层，它们之间既有联系又有区别。

客户端服务器模型（client-server model）是经典的网络应用程序模型，它分为服务端和客户端两个部分，这种分布式架构是现今大部分网络应用程序的基础，例如网站、网络游戏等都采用这种结构。

15.2.1 服务端

服务端（server）是一个这样的应用程序：它等待其他机器（即客户端）发送请求并回应它们。服务端的目的在于处理客户端的请求，在不同客户端间分享数据、软件或硬件资源。对于网络游戏来说，服务端的作用就是保存玩家信息，分享其他玩家的信息，实现玩家间的交互，处理战斗数据、怪物数据等。

15.2.2 客户端

客户端（client）或称为用户端，是指与服务器相对应，为客户提供本地服务的程序。除了一些只在本地运行的应用程序之外，一般安装在普通的客户机上，需要与服务端互相配合运行。目前，网络分为B/S（Brower/Server，即浏览器/服务器）与C/S（Client/Server，即客户端/服务器）两种架构，但是万维网使用的网页浏览器本身其实也是一个客户端。

游戏客户端展示游戏画面和游戏信息，提供给玩家交互，让玩家享受游戏体验的软件，玩家通过客户端输入指令，客户端将数据发送给服务器，服务器处理完给出结果，然后再由客户图形化出来。可以说，客户端是一个玩家和服务器之间的中介，使用Cocos2d-JS开发的游戏其实就是属于游戏客户端的范畴。

15.2.3 通信协议

通信协议（communication protocol），下面简称协议，是指双方实体（客户端或者服务端）完成通信或服务所必须遵循的规则和约定。协议定义了数据单元使用的格式、信息单元应该包含的信息与含义、连接方式、信息发送和接收的时序，从而确保网络中的数据顺利地传送到确定的地方，并且被正确地解析和处理。

目前，网络编程常用的协议有HTTP协议、TCP/IP协议和UDP协议这3种。

15.2.4 端口

端口是英文port的意译，可以认为是设备与外界通信的出口。端口可分为虚拟端口和物理端口，其中虚拟端口指计算机内部或交换机路由器内的端口，不可见，例如计算机中的80端口、21端口和23端口等。

每台设备在联网的情况下都会被绑定一个特定的IP地址，用于在网络世界中作为识别设备的唯一标识。如果把IP地址比作一间房子，端口就是出入这间房子的门。真正的房子只有几个门，但是一个IP地址的端口可以有65536（即2^{16}）个之多。端口是通过端口号来标记的，端口号只有整数，范围是从0到65535（即$2^{16}-1$）。

15.2.5 URI

统一资源标识符（Uniform Resource Identifier，URI）是一个用于标识某一互联网资源名称的字符串。该种标识允许用户对任何（包括本地和互联网）的资源通过特定的协议进行交互操作。URI由包括确定语法和相关协议的方案所定义。

URI一般由以下3部分组成。

- **主机名**：存放资源自身的名称，由路径表示。主机名称除了可以通过IP地址访问外，还可以是网站的域名访问，例如：http://www.baidu.com/res/img/testImage.png，它符合当前的RFC4395规范（协议名称://域名.根域名/目录/文件名.后缀）即可。
- **标识符**：有的URI指向一个资源的内部，这种URI以#结束，其后跟着一个anchor标志符（称为片段标志符），例如http://10.1.1.20:3000/a/b.php#a。其规范为“协议://域名/目录/文件#片段标识符”。
- **相对URI**：相对URI不包含任何命名规范信息，它的路径通常指同一台机器上的资源。相对URI可能含有相对路径（如“..”表示上一层路径），还可能包含片段标志符。

15.3　基于 HTTP 协议的通信

超文本传输协议（HTTP，HyperText Transfer Protocol）是互联网上应用最为广泛的一种网络协议，所有的WWW文件都必须遵守这个标准。设计HTTP最初的目的是为了提供一种发布和接收HTML页面的方法。

15.3.1　运作方式

HTTP协议的C/S（客户端/服务器）模式的信息交换过程是基于请求/响应范式的。它分为4个过程：建立连接、发送请求信息、发送响应信息、关闭连接。从这里可以看出网络连接的生命周期只存在于一次通信过程之中，一次通信结束后便会断开连接，这种连接方式称为“短连接”。

基于HTTP协议的请求主要分为get和post两种类型。

- **get请求**：一般用于获取服务端资源数据，并且安全要求相对较低的情况，比如获取HTML页面、JS文件等。同时，可以在请求URL字符串中将附加信息传递给服务端，但是由于URL的长度限制，附加的数据只能是微量的。
- **post请求**：一般用于发送数据到服务端，由于post请求发送的数据会包装成一个请求报文，请求报文中还会夹带请求本身的一些报文头等其他信息，是相对比较安全的一种请求方式。

运作时，游戏客户端要通过`HttpRequest`对象构建请求信息，然后创建一个远程连接，当连接成功时，便发送请求到指定的服务器，服务端会在一个特定的端口上面进行监听。当服务端监听到有HTTP请求到达时，会使用HTTP的协议规则对请求信息进行解析，然后进行相关的响应操作。之后服务端将处理的结果发送回游戏客户端。游戏客户端从`HttpResponse`对象中获取服务端的响应数据，然后进行相关的游戏逻辑处理，并关闭此次连接。

15.3.2　`XMLHttpRequest` 对象

在Cocos2d-JS中，发送HTTP网络请求时，需要用到`XMLHttpRequest`对象。`XMLHttpRequest`对象是Ajax的Web应用程序架构的一项关键功能，它可以同步或异步返回服务器的响应，并且能以文本的形式返回内容，你可以通过如下代码获得`XMLHttpRequest`对象：

```
1  var xhr = cc.loader.getXMLHttpRequest();
```

下面简要介绍`XMLHttpRequest`对象的常用方法。

1. open函数

`open`函数用于与服务器连接并创建一个新的请求对象，其函数接口如下：

```
1  open(method, url, async, user, password);
```

可以看到，该接口有5个参数，具体如下。

- **method**：String类型，表示请求类型，其值通常是GET或POST。
- **url**：String类型，表示要请求的服务端URL。
- **async**：Boolean类型的可选参数，表示是否使用异步连接，默认值为true。
- **user**：String类型的可选参数，如果请求需要用户名，可以通过该参数指定。
- **password**：String类型的可选参数，如果请求需要密码，可以通过该参数指定。

2. send()函数

打开连接之后，调用send()函数，就可以将构建好的请求发送到服务端。

3. abort()函数

abort()函数用于退出当前的请求。

除了上面的3个函数外，XMLHttpRequest还提供了如下属性。

1. readyState属性

readyState属性用来获取当前的状态，它确保服务器已经完成了一个请求。若readyState属性的值为4，则表示就绪状态，其状态如表15-1所示。

表15-1　readyState属性值的状态表

值	说　明
0	请求未初始化（还没有调用 open()）
1	请求已经建立，但是还没有发送（还没有调用send()）
2	请求已发送，正在处理中（通常，现在可以从响应中获取内容头）
3	请求在处理中，通常响应中已有部分数据可用了，但是服务器还没有完成响应的生成
4	响应已完成，可以获取并使用服务器的响应了

2. status属性

该属性用于获取当前HTTP请求的状态码，其中200表示请求成功，404表示未找到资源等，更多HTTP状态码读者可以自行上网搜索了解，或参考维基百科https://zh.wikipedia.org/wiki/HTTP状态码。

3. responseText属性

该属性可以用于获取服务器返回的响应文本。

4. onreadystatechange属性

该属性为Function类型的函数，用于设置请求的回调函数。当服务器处理状态改变后，会自动调用。

短连接的方式在手机应用中的使用非常广泛，这在iOS应用开发中几乎随处可见。而对于游戏应用，由于手机先天上的不足，手机端网游在通信方面也都设计得相对轻量，大多数情况下可能只是用于登录、发送积分和排行榜之类的数据。所以使用弱连接是很好的解决方案，能够将资源合理地利用起来。

下面通过一个示例，一起来学习下如何使用XMLHttpRequest对象进行通信，相关代码如下：

```
1  var xhr = cc.loader.getXMLHttpRequest();
2  // 链接服务器，url格式为： <URL>?xxx=xxx&yyy=yyy
3  xhr.open("GET", "http://httpbin.org/get?show_env=1", true);
4  // 请求发送
5  xhr.send();
6  // 回调函数
7  xhr.onreadystatechange = function () {
8      // 请求结束
9      if (xhr.readyState == 4 && (xhr.status >= 200 && xhr.status <= 207)) {
10         cc.log("Get请求成功！");
11         cc.log("请求状态码：", xhr.status);
12         cc.log("响应状态：", xhr.statusText);
13         cc.log("响应报文：", xhr.responseText);
14     }
15 };
```

在上面第3行代码中，我创建了一个GET请求，服务端使用httpbin.org进行测试，httpbin.org是一个HTTP客户端测试服务端，具体参见http://httpbin.org。然后在第5行设置了状态变化的回调函数。

第7行的判断用于筛选处理成功结束的状态变化，当条件成立时，即表示请求被成功处理了。

以上是发送GET请求的代码，要使用POST请求，只需要修改上述第3行代码即可：

```
1  xhr.open("POST","http://httpbin.org/post");
```

修改请求类型后，我们在发送数据的时候将数据传递给send()方法，相关代码如下：

```
1  xhr.send("This is message...(use POST)!");
```

运行上面的代码，可以在Chrome控制台看到GET请求和POST请求的返回结果如图15-2和图15-3所示。

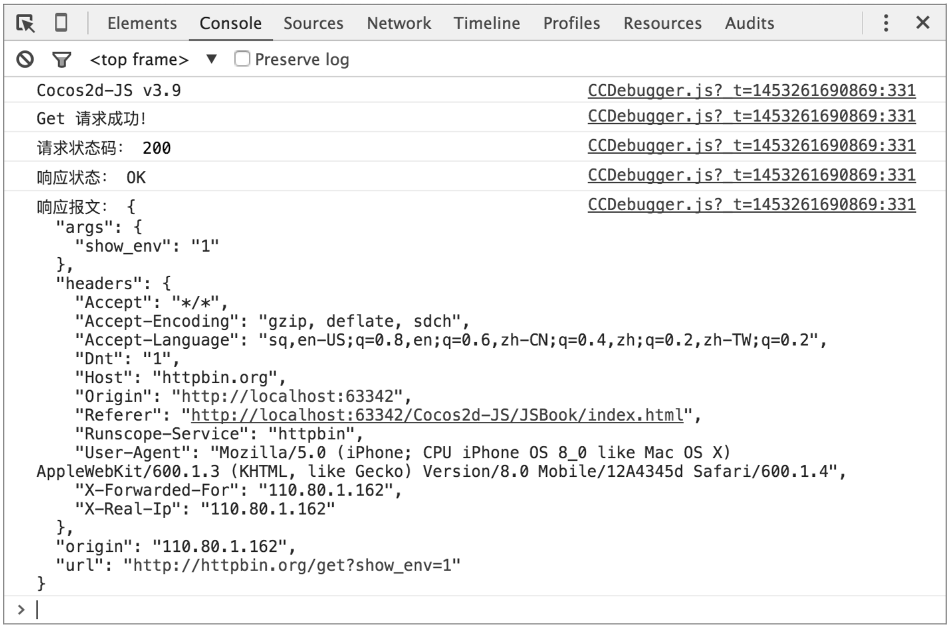


图15-2　控制台打印出`XMLHttpRequest`的返回信息

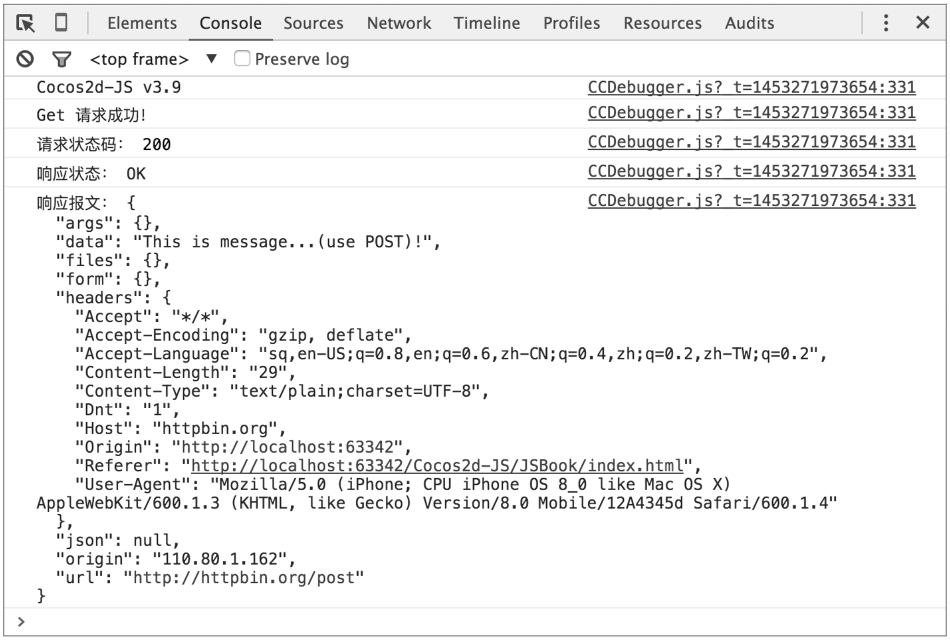


图15-3　控制台打印出`XMLHttpRequest`的POST返回信息

15.4　基于 WebSocket 的通信

虽然现在的移动端游戏对于网络的要求相对较低，但是在即时对战类游戏上，难免需要进行实时网络连接。

在浏览器中通过HTTP仅能实现单工通信，虽然comet（基于HTTP长连接的“服务器推”技术）可以在一定程度上模拟全双工通信，但效率较低，并且要求服务器有较好的支持。WebSocket协议是HTML5中一种新的协议，它不仅可以实现浏览器与服务器全双工通信（full-duplex），还能更好地节省服务器资源和带宽并达到实时通信的效果。

在WebSocket API中，浏览器和服务器只需要做一个握手的动作，然后浏览器和服务器之间就形成了一条快速通道，两者就可以直接互相传送数据。WebSocket互相沟通的header是很小的，大概只有2字节。

说明　在实现WebSocket连线的过程中，需要通过浏览器发出WebSocket连线请求，然后服务器发出回应，这个过程通常称为“握手”（handshaking）。另外，WebSocket协议本质上也是个TCP协议。

15.4.1　Cocos2d-JS 中的 WebSocket

由于浏览器的不同，其WebSocket的实现方式也不同，在Cocos2d-JS中可以使用如下代码获得WebSocket：

```
1  var WebSocket = WebSocket || window.WebSocket || window.MozWebSocket;
```

引入WebSocket之后，便可以创建WebSocket实例对象了，代码如下：

```
1  var webSocket = new WebSocket("ws://echo.websocket.org");
```

WebSocket的构造方法有两个参数，第一个参数是服务器的URL，第二个参数是协议类型（可省略）。`"ws://echo.websocket.org"`是websocket.org提供的测试服务端，URL中的`"ws://"`标识是WebSocket协议，与网页连接的`"http://"`相似。加密的WebSocket协议为`"wss://"`，类似于`"https://"`。

15.4.2　WebSocket 对象常用的属性和方法

使用WebSocket时，需要重写以下4个回调函数。

- **onopen(event)**：WebSocket在连接服务器时，如果连接成功，WebSocket就会调用`onopen`函数，告诉调用者客户端到服务器的通信链路已经成功建立，可以收发消息了。
- **onmessage(event)**：WebSocket在得到服务端的响应时，`onmessage`函数会被触发，该函数的参数`event`包含一个`data`属性，里面保存了服务端响应的内容。

- **onerror(event)**：客户端发送的请求，如果发生错误，就会收到onerror消息，游戏针对不同的错误码会做出相应的处理。
- **onclose(event)**：不管是服务器主动还是被动关闭了WebSocket，客户端在收到这个请求后，需要释放WebSocket内存，将WebSocket对象指向null。

在重写以上4个接口之后，便可以开始发送数据了。WebSocket对象的send()方法可以携带数据，将其发送往服务端，其代码如下：

```
1  webSocket.send("这是客户端发送的数据!");
```

但是在发送数据之前，应该保证WebSocket连接已经打开。通过WebSocket的readyState属性可以判断连接是否已经打开，相关代码如下：

```
1  if (webSocket.readyState == WebSocket.OPEN) {
2      cc.log("连接已打开，可以开始收发数据了。");
3      webSocket.send("这是客户端发送的数据!");
4  } else {
5      cc.log("连接未打开! ");
6  }
```

除了文本之外，WebSocket还可以发送二进制数据，其接口与发送文本一致，只是在数据操作上面需要做相应的处理。另外，值得注意的是，在不需要连接的时候一定要将其关闭，这可以通过WebSocket对象的close函数实现。下面演示一个示例。

首先，构建出WebSocket对象，链接测试服务端，并重写4个方法，相关代码如下：

```
1  var WebSocket = WebSocket || window.WebSocket || window.MozWebSocket;
2  var Unit15_NetworkLayer = cc.Layer.extend({
3      socket : null,
4      ctor : function(){
5          this._super();
6          this.loadWebSocket();
7      },
8      loadWebSocket : function(){
9          var self = this;
10         var socket = new WebSocket("ws://echo.websocket.org");
11         this.socket = socket;
12         socket.onopen = function(event) {    // 打开
13             cc.log("连接成功! 开始发送数据");
14         };
15         socket.onmessage = function(event) { // 响应
16             cc.log("收到响应数据: ", event.data);
17         };
18         socket.onerror = function(event) {   // 错误
19             cc.log("发生错误! ");
20         };
21         socket.onclose = function(event) {   // 关闭
22             cc.log("链接关闭! ");
23             self.socket = null;
24         };
25     }
26 });
```

由于网络进行握手连接需要一定的时间，所以我写了个延时发送数据的动作。这段代码我放在上面第24行代码的后面，具体如下：

```
1  loadWebSocket : function(){
2       // ......
3      socket.onclose = function(event) {    // 关闭
4          cc.log("链接关闭! ");
5        self.socket = null;
6      };
7
8      // 延迟一段时间之后发送数据
9      var delay = cc.delayTime(3);
10     var callFunc = cc.callFunc(function(){
11         if (socket.readyState == WebSocket.OPEN){
12             var strData = "这是客户端发送的字符码数据。";
13             var arrData = new Uint16Array(strData.length);
14             for (var i = 0; i < strData.length; i++) {
15                 arrData[i] = strData.charCodeAt(i);
16             }
17             socket.send(arrData.buffer);
18             cc.log("数据发送完毕! ");
19         } else {
20             cc.log("连接未打开! ");
21         }
22     }.bind(this));
23     this.runAction(cc.sequence(delay, callFunc));
24 }
```

在上述代码中，第12行到第16行代码通过一个for循环将数据转换成对应的字符码，然后保存在arrData数组里面。最后，第17行代码将数据发送到服务端。

发送的时候进行了字符转换，在接收的地方也要进行相应的操作。接下来，修改socket.onmessage函数内的逻辑代码，具体如下：

```
1  socket.onmessage = function(event) { // 响应
2      var arrData = new Uint16Array(event.data);
3      var data = "";
4      for (var i = 0; i < arrData.length; i++) {
5          if (arrData[i] == 0) {
6              data += "\'\\0\'";
7          } else {
8              var hexChar = "0x" + arrData[i].toString("16").toUpperCase();
9              data += String.fromCharCode(hexChar);
10         }
11     }
12     cc.log("收到响应数据: ", data);
13 };
```

但是运行修改后的代码，你会发现接收到的数据并没有正确显示出来。这是因为发送这种字符码数据，需要设置WebSocket的binaryType类型的值为arraybuffer，相关代码如下：

```
1  loadWebSocket : function(){
2       // ......
```

```
3       var socket  = new WebSocket("ws://echo.websocket.org");
4       this.socket = socket;
5       socket.binaryType = "arraybuffer";
6       // ......
7   };
```

运行实例，Chrome控制台的输出结果如图15-4所示。

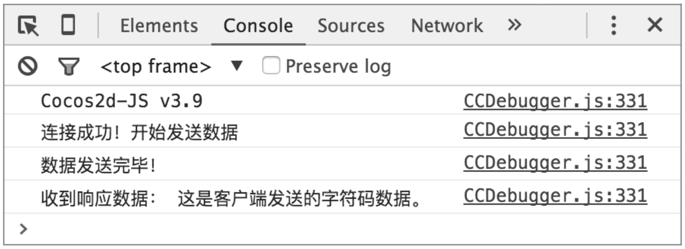


图15-4　WebSocket实例控制台的输出结果

15.5　基于 SocketIO 的通信

Node.js提供了高效的服务端运行环境，但是由于浏览器端对HTML5的支持不一，为了兼容所有的浏览器，提供卓越的实时的用户体验，并且为程序员提供客户端与服务端一致的编程体验，于是Socket.IO诞生了。

Socket.IO是一个开源的通信库，Node.js通过它可以实现WebSocket服务端，同时也提供客户端JS库。

Socket.IO支持以事件为基础的实时双向通信，它可以工作在任何平台、浏览器或移动设备上。Socket.IO解决了实时的通信问题，并统一了服务端与客户端的编程方式。启动了socket以后，就像建立了一条客户端与服务端的管道，两边可以互通有无。

Socket.IO支持4种协议——WebSocket、htmlfile、xhr-polling和jsonp-polling，它会自动根据浏览器选择适合的通信方式，从而让开发者可以聚焦到功能的实现而不是平台的兼容性上，同时Socket.IO具有不错的稳定性和性能。

由于Cocos2d-JS标准库不包含SocketIO网络模块，所以在使用SocketIO网络模块之前，应该在project.json文件的`modules`数组中添加`socketio`"值，相关代码如下：

```
1  "modules" : ["cocos2d", "socketio"],
```

15.5.1　获取 SocketIO 对象

首先，你需要获得SocketIO对象，代码如下：

```
1  var SocketIO = SocketIO || window.io;
```

得到SocketIO之后，可以通过SocketIO的`connect()`方法创建SocketIO的实例对象，并产生一个服务器连接。

`connect(uri, opts)`方法可以携带两个参数，第一个是URI路径字符串，第二个是一个操作数集合，示例代码如下：

```
1  var sioclient = SocketIO.connect("ws://tools.itharbors.com:4000", {"force new
   connection" : true});
```

其中ws://tools.itharbors.com:4000是可用于测试的URI。从URI中可以看出，Socket.IO在这里使用的也是WebSocket协议。

15.5.2　SocketIO 的常用方法

SocketIO对象包含以下几个常用方法。

- **`on(action, func)`**：该方法用于向SocketIO实例对象注册相关的动作（action）。这里的动作指的并非Cocos2d-JS中的`cc.Action`，而是一种JavaScript事件回调。第一个参数`action`是动作名称（事件名称），第二个参数`func`是对应的动作回调函数。
- **`send(msg)`**：该方法用于发送数据，它接收一个字符串作为参数。
- **`emit(action, args1, args2, ...)`**：该方法表示发送了一个`action`命令，命令是字符串类型的，在另一端接收时可以这么写：`socket.on('action',function (args1, args2){…})`。这里的`args`参数列表是可以传递`function`类型的参数，然后在动作回调里面进行调用。`action`的名称尽量使用全小写字母，否则可能无法被正确触发。
- **`disconnect()`**：该方法用于断开当前连接。网络连接相对还是比较消耗资源的，在不需要的时候应该断开它，比如玩家离线。

说明　使用SocketIO进行网络连接时，如果连接创建成功，会触发一个动作名称为`connect`的动作；关闭连接的时候，会触发`disconnect`动作；使用`send()`方法发送数据时，会触发`message`动作，该动作的回调函数应该携带一个参数，该参数表示发送的数据内容。

15.6　实例——使用 Node.js+Socket.IO 实现游戏聊天系统

在一些角色扮演类的游戏中，聊天功能可以说是必不可少的。在本章中，我们就一起来开发一个在线聊天系统，其中包含服务端和客户端的代码实现。服务端我打算用Node.js，这是因为我们可以直接使用JavaScript编写服务端代码。

在实例代码编写之前，不妨先来看看实例编写完成之后的效果，玩家吕布和貂蝉在此系统中的对话内容如图15-5和图15-6所示。



图15-5　吕布聊天窗口



图15-6　貂蝉聊天窗口

15.6.1　Node.js、Express 框架和 Socket.IO 库的安装

Node.js是一个采用C++语言编写而成的、开放源代码的、拥有跨平台特性，并且能够运行JavaScript代码的平台，它支持Mac OS X、Windows、Linux等操作系统。

Express是一套基于 Node.js 平台的快速、开放、极简的 Web 开发框架。

1. 在Mac OS X下安装

在Mac OS X下，我借助Homebrew工具安装Node.js，这意味着需要先安装Homebrew工具。该工具的安装较为简单，你可以在http://brew.sh/index.html中获取安装指令，如图15-7所示，然后在终端中运行指令即可。



图15-7　获取Homebrew

安装完Homebrew之后，便可在终端通过如下命令安装Node.js：

```
1  brew install node
```

接下来，再运行如下命令安装Express：

```
1  npm install express --g
```

安装完成之后，继续运行如下命令来安装Socket.IO：

```
1  npm install socket.io --g
```

全部安装完成后，在Mac OS X下，Node.js可以通过`node --version`命令获取版本号，从而验证是否安装成功，而Express和Socket.IO在安装完成后会直接给出反馈，如图15-8所示。

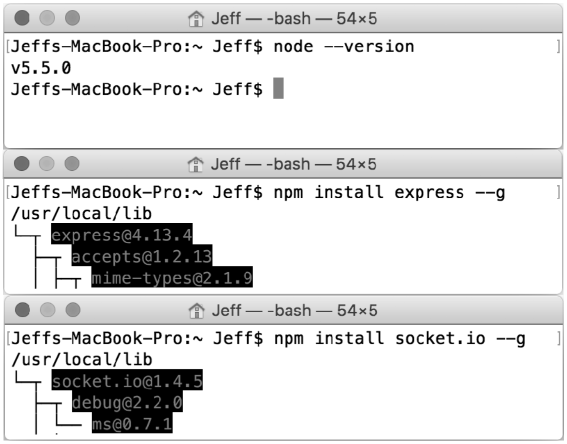


图15-8　Mac OS X下Node.js、Express以及socket.io的安装验证

2. 在Windows下安装

在Windows系统下，首先需要去Node.js官网上下载Node.js安装包，其下载地址为https://nodejs.org/download/。这里我下载的是node-v5.5.0-x64.msi安装包，双击打开它，然后一直点击Next按钮即可，如图15-9所示。

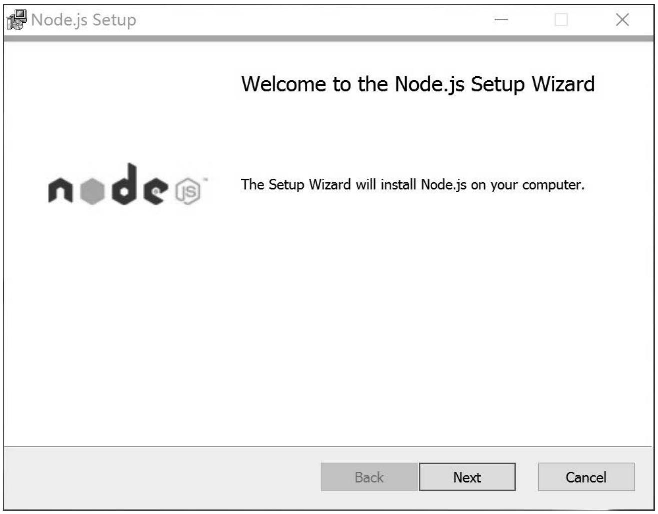


图15-9　Windows下的Node.js安装

同样，你也可以通过`node --version`指令获取到安装的Node.js版本。此外，Express和Socket.IO的安装指令与Mac OS X一致，代码如下：

```
1  npm install express --g
2  npm install socket.io --g
```

Windows下Node.js、Express以及Socket.io安装成功验证如图15-10所示。



图15-10　Windows下Node.js、Express以及Socket.IO的安装验证

15.6.2　服务端开发

当Node.js等安装完毕之后，便可以开始着手编写服务端代码了。此时我在桌面上新建一个Chat文件夹，然后在Chat文件夹中新建一个名字为server的JavaScript文件，最后在WebStorm中打开Chat文件夹，如图15-11所示。

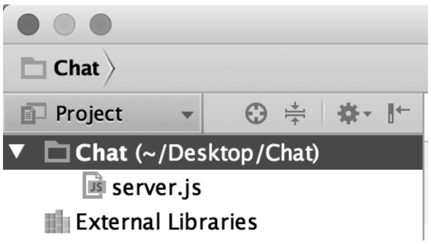


图15-11　Chat文件夹的目录结构

在server.js文件中，首先引入3个模块，分别是`express`、`http`以及`socket.io`，然后声明两个变量来保存当前在线玩家信息以及在线人数，代码如下：

```
1  var app = require("express")();           // 引入express模块
2  var http =  require("http").Server(app); // 引入http模块
3  var io = require("socket.io")(http);     // 引socket.io模块
4  var onlineplayers = {};      // 在线玩家信息
5  var onlineCount = 0;         // 当前在线人数
```

15

说明　本章通过在线聊天系统实例来巩固之前所学的知识点，但服务端知识并不是本章以及本书的重点，若你对本实例的服务端代码不理解，也是无妨的，大概懂它是干嘛的即可。

然后，再编写一段代码用来监听客户端的连接以及3000端口，具体如下：

```
1  // 监听连接
2  io.on("connection", function (socket) {
3      console.log("一个用户连接了！");
4  });
5
6  http.listen(3000, function(){
7      console.log("listening on : 3000");
8  });
```

此时，打开终端（Windows对应命令行工具），通过如下命令来启动服务端：

```
1  node /Users/Jeff/Desktop/Chat/server.js
```

启动成功之后，可以看到如图15-12所示的反馈。

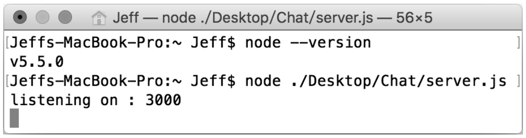


图15-12　启动服务端成功

此时虽然服务端已经启动成功，但是我还并未开始编写玩家登录、聊天等逻辑代码，接下来就来实现它们。

玩家的登录以及聊天等功能应当放在连接之后，所以登录和聊天等逻辑代码我放在`io.on`函数中实现，具体如下：

```
1  // 监听连接
2  io.on("connect", function (socket) {
3      console.log("一个用户连接了！");
4
5      // 监听玩家[登录]
6      socket.on("login", function(obj){
7          var data = JSON.parse(obj); // 从字符串中解析出JSON对象
8          // 将新加入用户的唯一标识当作socket的名称，后面退出的时候会用到
9          socket.name = data.playerId;
10         // 检查在线列表，如果不在里面就加入
11         if(!onlineplayers.hasOwnProperty(data.playerId)) {
12             onlineplayers[data.playerId] = data.playerName;
13             onlineCount++;  //在线人数+1
14         }
15         var loginData = {onlineplayers : onlineplayers, onlineCount : onlineCount,
           player : data};
16         // 向所有客户端广播用户加入
```

```
17          io.emit("login", JSON.stringify(loginData));
18         console.log("【" + data.playerName + "】进入了游戏! ");
19     });
20
21    // 监听玩家发布聊天内容
22     socket.on("message", function(obj){
23         // 向所有客户端广播发布的消息
24         io.emit("message", obj);
25         var data = JSON.parse(obj);
26         console.log(data.playerName + "说: " + data.content);
27     });
28
29    // 监听玩家[退出]
30     socket.on("disconnect", function(){
31         // 将退出的用户从在线列表中删除
32         if(onlineplayers.hasOwnProperty(socket.name)) {
33             // 退出玩家的信息
34             var playerData = { playerId: socket.name, playerName : onlineplayers
               [socket.name]};
35             delete onlineplayers[socket.name];  // 删除玩家
36             onlineCount--;  // 在线人数-1
37
38             var logoutData = {onlineplayers : onlineplayers, onlineCount :
               onlineCount, player : playerData};
39             // 向所有客户端广播用户退出
40             io.emit("logout", JSON.stringify(logoutData));
41             console.log(playerData.playerName + "退出了游戏");
42         }
43     });
44 });
```

需要说明的是，第7行代码从字符串中解析出JSON对象。虽然在Web中数据的发送和接收支持`Number`类型，但是在JSB上仅支持字符串形式，所以在数据发送之前，应当通过`JSON.stringify(obj)`方法将JSON对象转为字符串，然后发送到服务端。

至此，服务端代码编写完毕，其中你需要注意的是一些事件名称，例如`login`、`message`以及`logout`等，这些必须和客户端一一对应。

15.6.3 客户端开发

从图15-5可以看出，客户端需要三个Text来显示玩家ID、玩家名字和在线人数，中间聊天记录面板为一个ListView，其中的对话内容用Text表示，而在ListView下面则是一个TextField编辑框和一个“发送”按钮。我相信你看完第10章之后，开发这些UI控件已不在话下，所以这里我不打算贴出UI的实现代码，你可以自己编写或者打开随书源码进行阅读。

这里重点来看一下通信部分，代码如下：

```
1  loadSocketIO : function(){
2      var socketIO = SocketIO.connect('127.0.0.1:3000' );
3     this.socketIO = socketIO;   // 保存socketIO
```

```
4
5       var self = this;
6       socketIO.on("connect", function(data){
7           cc.log("连接服务器成功...");
8           // 发送[登录请求]
9          socketIO.emit('login', JSON.stringify(self.playerData));
10
11          // 监听[玩家登录]
12          socketIO.on('login', function(obj){
13              var data = JSON.parse(obj);
14              self.isLogin = true;
15              self.onlineCountText.setString("在线人数：" + data.onlineCount);
16              cc.log("玩家【" + data.player.playerName + "】进入游戏了！");
17          });
18
19          // 监听[玩家聊天]
20          socketIO.on('message', function(obj){
21              var data = JSON.parse(obj);
22              var string = data.playerName + "[" + data.playerId + "] 说：" +
                data.content;
23              self.applyMessageText(string);
24          });
25
26         // 监听[玩家退出]
27          socketIO.on('logout', function(obj){
28              var data = JSON.parse(obj);
29              self.onlineCountText.setString("在线人数：" + data.onlineCount);
30              cc.log("玩家【" + data.player.playerName + "】退出游戏！");
31          });
32      });
33  }
```

第2行代码创建了一个连接，127.0.0.1是服务器监听地址，这里用本地IP。第6行监听连接状态，当连接创建成功时，便可以发送登录（login）请求，如第9行代码所示。玩家登录后，服务端会通知其他玩家，那客户端自然也需要实现玩家登录监听，就像第12行代码所示。当有玩家上线后，客户端需要更新当前在线人数以及在线玩家的信息，而这些服务端都发送过来了。在第13行代码中，我将服务端发送过来的数据转为JSON对象。在第15行代码中，更新了当前服务器在线人数的文本显示。

玩家之间的聊天实现和登录功能类似，第20行代码实现了聊天信息的监听，第22行代码做了一个字符串处理，第23行代码将处理之后的字符串创建为一个Text对象，并添加到ListView中，实现代码如下：

```
1  // 添加信息文本到ListView
2  applyMessageText : function(string){
3      var layout = new ccui.Layout();
4      this.listView.pushBackCustomItem(layout);
5
6      var text = new ccui.Text(string, "", 18);
7      layout.setContentSize(text.getContentSize());
8      text.setPosition(layout.width / 2, layout.height / 2);
```

```
9      layout.addChild(text);
10 }
```

在发送消息功能上，逻辑大概是这样的：玩家在TextField编辑框中输入要发送的信息，然后点击“发送”按钮，客户端调用socketIO.emit方法将信息发送出去。在这个过程中，首先需要获取TextField编辑框中的内容，然后在“发送”按钮松开后将消息发送出去，代码如下：

```
1  // 加载[“发送”按钮]
2  loadSendButton : function(){
3      var fileName = "res/unit10_ui/scale9.png";
4      var node = new ccui.Button();
5      this.addChild(node);
6     // ......
7      node.addTouchEventListener(function(sender, type) {
8          switch (type) {
9              case ccui.Widget.TOUCH_ENDED:
10                 var message = this.inputText.getString();
11                 this.onSendMessage(message);
12                 this.inputText.setString(""); // 清空编辑框内容
13             break;
14         }
15     }.bind(this));
16 }
```

第10行代码获取TextField中所输入的内容，第11行代码将信息发送出去，代码如下：

```
1  // 发送[信息]
2  onSendMessage : function(message){
3      if (!this.isLogin){ cc.log("没有登录"); return }
4     if (!message) { return; }   // 空字符串则不发送
5
6      var messageData = {
7         playerId : this.playerData.playerId,
8          playerName : this.playerData.playerName,
9          content : message
10     };
11     // 发送[聊天请求]
12     this.socketIO.emit('message', JSON.stringify(messageData));
13 }
```

至此，整个聊天系统开发完毕，启动服务端之后，运行多个客户端，便可实现聊天。当有玩家登录，或者发送信息等，终端显示的日志如图15-13所示。



图15-13　服务器终端显示

15.7　小结

通过本章的学习，我们学习了网络编程相关的基本概念以及`XMLHttpRequest`、WebSocket和SocketIO的用法。最后，在本章的实例中，实现了一个Node.js + Socket.io的在线聊天系统。

15.8　参考资源

本章的参考资源为Node.js：https://zh.wikipedia.org/wiki/Node.js。

第 16 章 JavaScript Binding

纵观世上的计算机主流语言，其种类繁多，每种语言都有自己的优势和缺陷。编译型语言虽然拥有较好的性能，但是编写复杂。解释型语言虽然灵活，但是性能往往稍有缺憾。在游戏开发中，C++往往以其卓越的性能和更接近底层的特性成为编写游戏引擎的首选。然而C++虽拥有超高的性能优势，但相对脚本语言而言，它显得过于复杂和难于上手，开发门槛高，开发周期长，而且很容易因为团队成员水平的参差不齐和编程风格的迥异造成团队配合上的困难。还有最重要的一点，那就是C++不支持热更新①，这并不适合目前国内的手游环境，所以往往在游戏技术选型阶段就被排除在外。

所幸的是，Mozilla给我们带来了解决方案，那就是SpiderMonkey。SpiderMonkey是Mozilla开发的JavaScript引擎，使用C/C++编写，用在多个Mozilla产品中，包括Firefox、Thunkerbird等。SpiderMonkey为Cocos2d-x提供了C++调用JavaScript的能力，使得Cocos2d-x能够跟JavaScript结合起来，开发者便可以使用JavaScript来编写游戏的业务逻辑代码，同时又不失去底层C++引擎框架的高性能。更妙的是，它使我们可以使用同一份游戏代码，不经修改就可以跑在浏览器和其他平台上，这大大地拓宽了游戏的运行场景和业务范围。

但是，这个过程并不是自动的。要用SpiderMonkey实现C++调用JavaScript，我们需要为其编写大量的胶水层代码（即两种语言可以相互调用的中间代码），这份工作不仅烦琐、枯燥，而且工作量大，容易出错。此外，一旦Cocos2d-x的接口有些许变化，相应的绑定代码就得跟着改变，给维护工作带来了相当大的困难。

为了解决上述问题，时任Zygna工程师的Rolando Abarca于2012年主导并开发了一套基于Cocos2d-x和SpiderMonkey的JavaScript自动绑定技术，它让我们可以通过一些简单的配置和少量的代码生成绝大部分的胶水层代码，减少了绑定上的工作量，也大大改善了维护难度。

JavaScript Binding一直是Cocos2d-JS团队所引以为豪的核心技术所在，本章将围绕该技术展开，详解其背后的原理，并一步一步引导读者如何绑定自己的C++类到JavaScript中。

① 热更新是指不需要重新打包发布到各个渠道包，直接在游戏开启时判断版本号，然后从服务端拉取代码和资源进行更新操作。

本章内容：

- ❑ Cocos引擎中的脚本绑定框架
- ❑ 自动绑定
- ❑ 深入探究绑定技术
- ❑ 绑定自己的C++类
- ❑ 实例——绑定SQLite3

16.1　Cocos 引擎中的脚本绑定框架

众所周知，Cocos游戏引擎共有3个主流分支：C++、JavaScript和Lua。其中，C++为核心框架，JavaScript和Lua通过脚本绑定技术架设在Cocos2d-x之上。图16-1为Cocos引擎脚本绑定框架示意图。

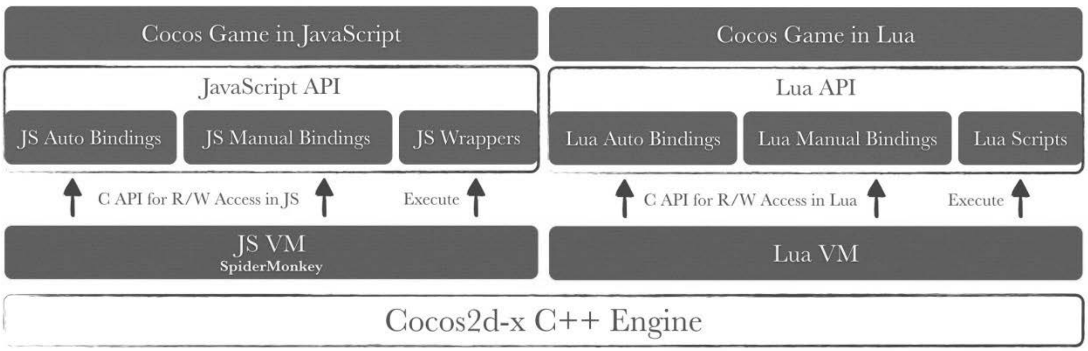


图16-1　Cocos引擎脚本绑定框架示意图

如图16-1所示，可将绑定框架分为4层，最底层为C++编写的Cocos2d-x游戏引擎，它将自己实现的功能暴露出去，供上层的脚本语言引擎调用。

往上一层，是JavaScript和Lua的脚本引擎，它用来解析JavaScript和Lua文件。其中，JavaScript绑定使用的是SpiderMonkey引擎，它由Mozilla维护。SpiderMonkey是世界上第一个JavaScript引擎，也是目前JavaScript执行性能最优秀的引擎之一。脚本引擎提供了两个重要的能力：其一是执行脚本的能力，它让C++环境能够像浏览器一样去解析并执行JavaScript语言，其二为它提供了对脚本层进行访问的C API。毫无疑问，对脚本层访问的能力是底层Cocos2d-x引擎到脚本层的重要桥梁，它让C++函数可以获取到脚本的API、访问脚本环境中的变量、构造新的对象、主动调用脚本层的函数等。

再往上一层，是脚本绑定层，即胶水层，在Cocos2d-JS中称为Cocos2d-x JSB，它是JavaScript到Cocos2d-x引擎的通道，这一层由三部分组成：自动绑定、手动绑定和纯脚本实现。纯脚本实现用来改造一些绑定之后的API，并且提供一些不需要绑定的API，使其更适合脚本程序员的习惯，

让API变得更加灵活、简单。例如，`cc.director`、`cc.textureCache`等就是对`cc.Director-.getInstance()`、`cc.director.getTextureCache()`的改造。

而最上层，则为脚本游戏层，它通过绑定等一系列技术，使得开发者可以使用JavaScript或Lua来编写游戏业务逻辑代码，让游戏逻辑开发绕过了笨重的编译性语言，过渡到脚本语言上，给开发者带来了更简单、更快速的体验。

16.2 自动绑定

在Cocos2d-x引擎的GitHub仓库里，存在一位神秘的代码贡献者，它的神秘在于外界少有人知，它就是CocosRobot（机器人），如图16-2和图16-3所示。实际上，CocosRobot是Cocos引擎团队放在Travis CI上面的一个构建任务，每当Cocos仓库有代码合并的时候，就会触发这个任务，此任务会将引擎组人员提交上来的需要绑定的代码进行自动绑定，然后再提交到GitHub上。

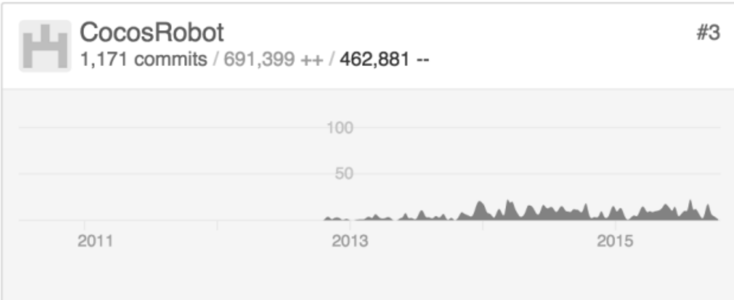


图16-2 CocosRobot账号

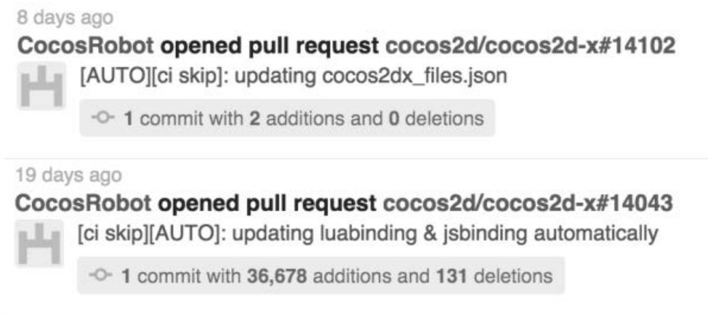


图16-3 CocosRobot代码提交记录

CocosRobot的根基就是自动绑定技术。这里自动绑定技术不仅完成了Cocos引擎海量的API自动绑定，还解决了每个Cocos引擎版本迭代所带来的脚本绑定影响，极大地提高了引擎绑定模块的可维护性和灵活性。

除了CocosRobot和引擎团队之外，开发者也可以自由使用自动绑定工具绑定自己编写的C++类。自动绑定技术降低了绑定成本，使得一个不懂绑定技术的开发人员都可以通过自动绑定技术将自己的C++类绑定到脚本层。

16.2.1　自动绑定简介及其限制

Cocos引擎中的脚本自动绑定技术得益于Bindings Generator，它是一个使用Python语言编写的脚本工具，位于cocos2d-x/tools/bindings-generator目录下。

虽然自动绑定工具很强大，但它有以下几个限制。

- 只能够针对类生成绑定，不可以绑定结构体、枚举类型、宏定义、宏函数等。
- 不能够生成纯虚类的API，因为自动绑定无法实例化纯虚函数对象，在绑定层维护它。
- 如果子类中重写了父类API的同时，又重载了这个API，则此API将无法被绑定。
- 无法绑定API的参数或返回值使用了没有被自动绑定绑定的类或者结构体。
- 无法自动绑定实现C++运行时的回调函数。

因为自动绑定工具具有一定的局限性，绑定出来的结果可能会导致编译上或行为上的错误，所以碰到上面的情况时，需要手动绑定。

说明　因为手动绑定将展开非常庞大的知识体系，所以本书暂不讲解。

16.2.2　配置绑定环境

前面提到过，绑定框架中有4层，最底层的C++类、JavaScript和Lua的脚本引擎、中间的脚本绑定层以及最上层的游戏脚本逻辑。脚本绑定层中的绑定代码是绑定工具自动生成的，绑定工具需要依赖其他一些环境才能生成绑定代码。接下来，就来看一下如何在Mac OS X以及Windows操作系统中配置绑定环境。

1. Mac OS X下的环境配置

在Mac OS X上，绑定环境的配置较为简单，大部分的步骤可以利用终端命令完成，具体有如下5步。

(1) 安装Python 2.7：Mac OS X系统本身自带Python，此处应保证Python版本为2.7.x，具体可参见本书第2章。

(2) 安装Pip：终端键入 `sudo easy_install pip`后回车。

(3) 通过Pip安装Python依赖库PyYAML：终端键入`sudo pip install PyYAML`后回车。

(4) 通过Pip安装Python依赖库Cheetah：终端键入`sudo pip install Cheetah`后回车。

(5) 下载Android NDK r9d：具体可参见第2章。

在Mac OS X上，PyYAML和Cheetah的成功安装示意图如图16-4和图16-5所示。

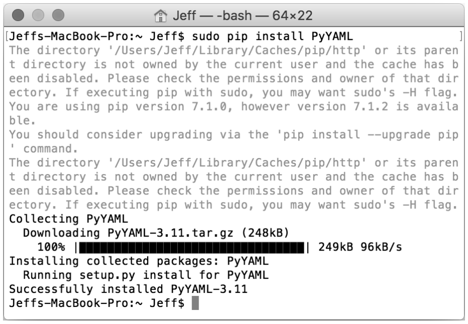


图16-4 PyYAML安装成功示意图（Mac OS X）

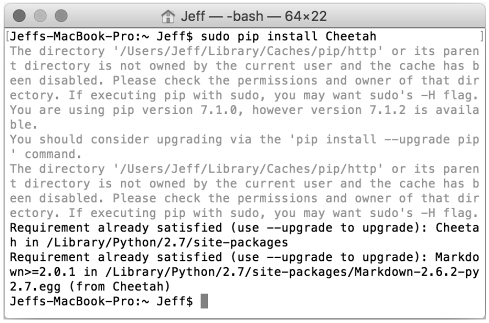


图16-5 Cheetah安装成功示意图（Mac OS X）

2. Windows下的环境配置

相比Mac OS X，Windows中的绑定环境配置也大同小异，具体步骤如下。

(1) 安装Python 2.7.3（32位）：下载32位的Python 2.7.3版本，双击安装包开始安装，安装完成后一定记得设置环境变量（参见本书第2章）。其下载地址：http://www.python.org/ftp/python/2.7.3/python-2.7.3.msi。

(2) 安装Python依赖库PyYAML：PyYAML的安装与Python大同小异，只需一直点击“下一步”即可。其下载地址：http://pyyaml.org/download/pyyaml/PyYAML-3.10.win32-py2.7.exe。

(3) 下载Python依赖库Cheetah：下载好后，解压到Python安装路径下的Lib\site-packages文件夹中即可。下载地址：https://raw.github.com/dumganhar/my_old_cocos2d-x_backup/download/downloads/Cheetah.zip。

(4) 下载Android NDK r9d：参见第2章。

在Windows上，PyYAML和Cheetah的成功安装示意图如图16-6和图16-7所示。

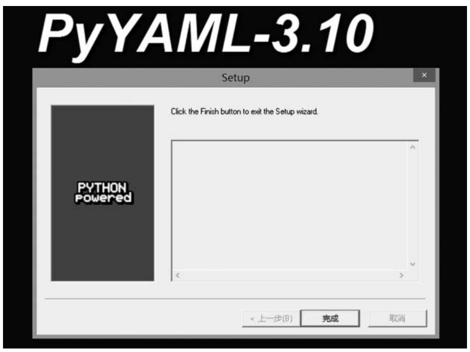


图16-6 PyYAML安装成功示意图（Windows）

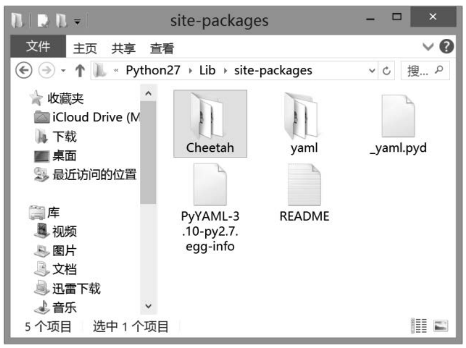


图16-7 Cheetah安装成功示意图（Windows）

3. 验证环境是否配置成功

自动绑定需要的环境配置完毕后，运行引擎包下的tools/tojs/genbindings.py脚本，若出现如图16-8所示的界面（Windows上的界面如图16-9所示），则说明环境已经配置成功。

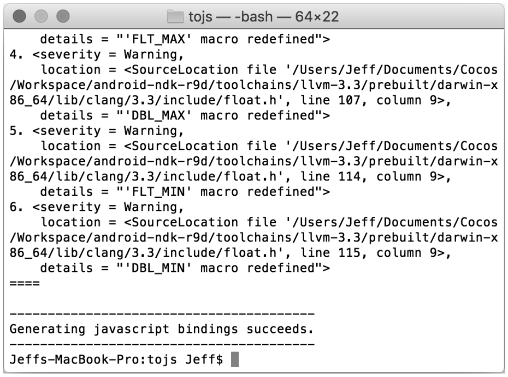


图16-8　Mac OS X上环境配置成功验证

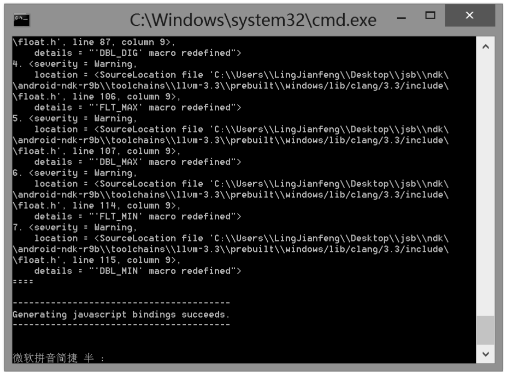


图16-9　Windows上环境配置成功验证

需要特别说明的是，在Mac OS X操作系统中，且你的系统版本大于10.11，那么在生成绑定代码的时候，若报一个libclang.dylib找不到的错误（`libclangError: dlopen(libclang.dylib,6): image not found.`），则是因为Mac OS X 10.11系统添加了一个称为System Integrity Protection (SIP)系统完整性保护的功能，我们需要先关闭它，具体方法如下。

(1) 开机按住command+ R键，进入恢复模式。

(2) 打开“使用工具”→“终端”。

(3) 输入命令`csrutil disable`来关闭。同样，打开命令为：`csrutil enable`。

(4) 关闭成功之后，重启进入系统即可。

16.3　深入探究绑定技术

绑定环境配置成功之后，即可深入了解绑定技术了。绑定技术的原理是通过libclang分析C++头文件，以一定的绑定规则和绑定代码模板，根据配置文件的配置，为对应的C++类公有方法一一生成绑定代码，从而将C++的API暴露给脚本层，供脚本调用。

16.3.1　函数注册

函数注册指的是将C++类注册进JavaScript环境中。绑定工具会自动生成绑定代码，其中包含一个.hpp文件和一个.cpp文件，.hpp文件包含函数注册方法的声明，而.cpp文件则是对函数注册方法的实现。Cocos引擎已经自动绑定好的代码位于cocos2d-x/cocos/scripting/js-bindings/auto文件夹下。

这里以Node类的setOpacity函数为例，看看它是被如何注册进JavaScript环境中的。

首先，在AppDelegate类的applicationDidFinishLaunching函数中可以看到如下代码：

```
1  bool AppDelegate::applicationDidFinishLaunching(){
2      // ......
3      ScriptingCore * sc = ScriptingCore::getInstance();// 脚本引擎单例对象
4      sc->addRegisterCallback(register_all_cocos2dx); // 注册函数
5      // ......
6  }
```

在上面的代码中，ScriptingCore为脚本引擎类，它是一个单例类。第4行代码将register_all_cocos2dx方法注册到脚本引擎中。register_all_cocos2dx方法在cocos2d-x/cocos/scripting/js-bindings/auto/jsb_cocos2dx_auto.cpp文件里，可以在这个方法里看到如下代码：

```
1  void register_all_cocos2dx(JSContext* cx, JS::HandleObject obj) {
2      // 获取JavaScript层的命名空间
3      JS::RootedObject ns(cx);
4      get_or_create_js_obj(cx, obj, "cc", &ns);
5      // ......
6      js_register_cocos2dx_Node(cx, ns);
7      // ......
8  }
```

上述代码第6行为类绑定函数。在js_register_cocos2dx_Node函数中，它实现了类成员的绑定。而Node的setOpacity函数就是通过下面的方式生成了JavaScript的API，并且绑定到了C++类的函数：

```
1  void js_register_cocos2dx_Node(JSContext *cx, JS::HandleObject global) {
2      jsb_cocos2d_Node_class = (JSClass *)calloc(1, sizeof(JSClass));
3      jsb_cocos2d_Node_class->name = "Node";
4      // ......
5      static JSFunctionSpec funcs[] = {
6          // ......
7          JS_FN("setOpacity", js_cocos2dx_Node_setOpacity, 1, JSPROP_PERMANENT |
           JSPROP_ENUMERATE)
8          // ......
9          JS_FS_END
10     };
11     // ......
12 }
```

上述为API绑定函数。在第7行代码中，宏函数JS_FN的第一个字符串参数"setOpacity"是JSB的API，它供脚本层调用。我们可以改动它，但是一般不建议这样做，因为这样会破坏了JSB引擎和HTML5引擎API的一致性。

第二个参数为C++层的对应函数。例如，SpiderMonkey调用JavaScript层中的node.setOpacity()函数时，在Native上最终调用到js_cocos2dx_Node_setOpacity函数，而该函数为Node类setOpacity方法的绑定实现。

第三个参数是函数调用时的参数个数。

最后一个参数是一些访问特性，JSPROP_PERMANENT表示不可删除，JSPROP_ENUMERATE表示在枚举时可见。

下列代码为js_cocos2dx_Node_setOpacity函数的实现：

```
1  bool js_cocos2dx_Node_setOpacity(JSContext *cx, uint32_t argc, jsval *vp){
2      // 1. 通过SpiderMonkey API获取脚本层传入的所有参数
3      JS::CallArgs args = JS::CallArgsFromVp(argc, vp);
4      bool ok = true;
5      // 2. 通过SpiderMonkey API获取本次调用的脚本层调用者对象
6      JS::RootedObject obj(cx, args.thisv().toObjectOrNull());
8      js_proxy_t *proxy = jsb_get_js_proxy(obj);
9      // 3. 转换脚本层调用对象为其对应的C++对象
10     cocos2d::Node* cobj = (cocos2d::Node *)(proxy ? proxy->ptr : NULL);
11     JSB_PRECONDITION2( cobj, cx, false, "js_cocos2dx_Node_setOpacity : Invalid
       Native Object");
12     if (argc == 1) {
13         uint16_t arg0;
14         // 4. 转换脚本层传入的参数对象为C++值或对象
15         ok &= jsval_to_uint16(cx, args.get(0), &arg0);
16         JSB_PRECONDITION2(ok, cx, false, "js_cocos2dx_Node_setOpacity : Error
           processing arguments");
17         // 5. 最终使用转换过的参数调用实际的C++ API
18         cobj->setOpacity(arg0);
19         // 6. 设置脚本层函数调用的返回值
20         args.rval().setUndefined();
21         return true;
22     }
23
24     JS_ReportError(cx, "js_cocos2dx_Node_setOpacity : wrong number of arguments:
       %d, was expecting %d", argc, 1);
25     return false;
26 }
```

上述代码完成了Node类中setOpacity方法的绑定，可以在这里做一个小结，将上述的几个步骤归纳一下。

脚本自动绑定工具绑定出来的结果是一个.hpp头文件和一个.cpp实现文件，这两个文件完成如下几个功能。

(1) 完成Cocos2d-x引擎所有的C++ API的绑定函数，用于桥接脚本环境中的API和C++ API。在脚本层调用相应API时，实际调用的是绑定函数，绑定函数最终转发调用C++ API，如js_cocos2dx_Node_setOpacity函数。

(2) 完成C++类的绑定函数，用于在脚本环境中创建对应的类，它会将所有API的绑定函数注册到脚本类中，这样在脚本中调用这些API时就会调用到绑定函数，例如js_register_cocos2dx_Node函数。

(3) 注册绑定函数，这个函数会调用C++类的绑定函数。调用这个注册函数，会让这些C++类被实际注册到脚本环境中，例如：sc->addRegisterCallback(register_all_cocos2dx)

等。整个过程如图16-10和图16-11所示。

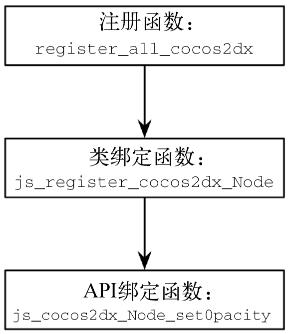


图16-10　绑定注册过程

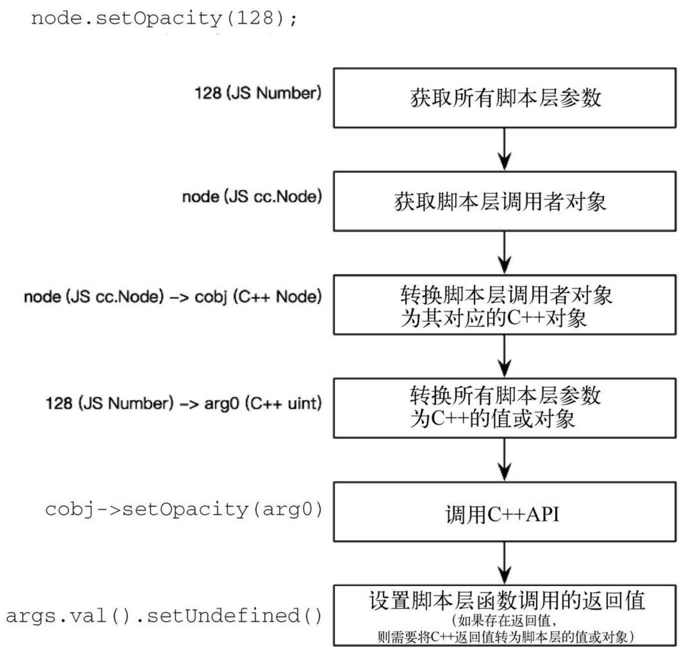


图16-11　JSB调用过程

另外，cocos2d-x/cocos/scripting/js-bindings/auto/api文件夹里面的JavaScript文件为绑定时自动生成的API，它仅仅只是一个说明文件，通常被用来生成文档，或者作为其他IDE中代码补全的基础。例如，可以提取api文件夹里的JavaScript文件，然后根据Sublime的代码补全规则写一个脚本，从而生成Sublime编辑器的代码补全。

16.3.2　分析 C++头文件

Bindings Generator自动绑定工具通过libclang的Python API对C++头文件进行语法分析，然后对C++类的API一一生成绑定代码，其绑定的整个过程如下（真实代码处理过程与下列步骤的顺序并不严格一致，代码的处理过程是边扫描边读取配置边生成）。

(1) 根据tools/tojs/genbindings.py脚本的代码识别和处理相关的系统环境。

(2) 根据tools/tojs/genbindings.py脚本传入脚本绑定需要的参数，包括相关的配置文件和对应的需要生成的目标文件的目录。

(3) 运行tools/bindings-generator/generator.py脚本，根据第(2)步传入的配置文件找到相关C++代码目录，扫描相关的头文件。

(4) 根据第(2)步传入的配置文件里的规则确定生成、跳过、重命名等转换规则。

(5) 根据tools/bindings-generator/targets/spidermonkey/文件夹里的模板文件和conversions.yaml配置文件确定代码的转换规则。

(6) 生成C++类的API绑定代码并输出到目标文件中。

16.3.3　绑定规则

绑定规则是一个以.ini为后缀的文本配置文件，例如cocos2d-x/tools/tojs/下的cocos2dx.ini、cocos2dx_ui.ini等，它对应的是16.3.2节第(2)步所读入的配置。通常情况下，我们添加的大部分新的C++ API接口自动绑定配置都在这里，其主要部分如表16-1所示。

表16-1　自动绑定配置信息说明

参　数	说　明
prefix	作为自动绑定工具生成的绑定函数的函数名前缀
target_namespace	脚本中的目标命名空间，比如cc、ccui、spine等
clang_flags	clang标签，其中可以添加预编译宏
macro_judgement	提供了自动绑定的预编译控制，我们可以通过预编译选项有条件地导出相应的API，例如平台的控制和debug的控制等
headers	需要被绑定的头文件列表，以空格分隔，头文件将被扫描，同时也会扫描头文件中包含的头文件
classes	需要被绑定的类名列表，以空格分隔
classes_need_extend	需要在脚本层被继承的类列表，以空格分隔
skip	需要忽略的API列表，格式为ClassName::[api1 api2]，不同的类以逗号分隔
rename_functions	需要被重命名的函数，会将C++中的函数绑定为指定名字的脚本函数，格式为ClassName::[cppFunctionName=scriptFunctionName ...]，不同的类以逗号分隔
rename_classes	需要被重命名的类，会将C++中的类名绑定为指定的脚本类名，格式为CppClassName::ScriptClassName，以逗号分割
classes_have_no_parents	没有父类的类列表，以空格分隔
abstract_classes	没有构造函数的类列表，以空格分隔

16.3.4　绑定模板

有了这些配置后，绑定工具是否就可以生成各种API的绑定函数呢？答案是否定的，正所谓一个巴掌拍不响。现在绑定工具虽然知道了哪些API要被绑定，但是它并不知道要以什么样的方式进行绑定，所以还需要配合各种API的绑定代码模板才可生成各种API的绑定函数。对于每一种特定的模板，自动绑定工具会读取clang.cindex解析出来的类或者API定义信息以及绑定配置信息，从而生成特定API的绑定代码。这些模板放在cocos2d-x/tools/bindings-generator/targets/spidermonkey/templates下，大概有以下几种：

- 头文件和cpp文件的头部代码模板
- 头文件和cpp文件的尾部代码模板
- 头文件内容模板，包含脚本层类对象声明、原型对象声明和API绑定函数声明
- 类绑定函数模板
- 构造函数的绑定函数模板
- 静态函数的绑定函数模板
- 重载的静态函数的绑定函数模板
- 公共属性的绑定函数模板
- 公共方法的绑定函数模板
- 重载的公共方法的绑定函数模板
- lambda函数的绑定函数模板

16.3.5 转换函数

自动绑定工具虽然强大，但是它毕竟仅仅只是个工具，在一些问题上还是需要我们参与进来的，例如我们需要告诉动绑定工具如何在各种C++类型和脚本类型之间转换。

Cocos2d-x引擎中C++和JavaScript对象的互相转换函数规则在cocos2d-x/tools/bindings-generator/targets/spidermonkey/conversions.yaml文件中，其中包含的转换有以下几点。

- **本地数据类型转换**：在自动绑定中，C++的`short`类型转换成`int32_t`类型，`float`类型被转换成`double`类型，这是因为`float`类型没有办法生成绑定。代码见conversions.yaml文件中的`native_types`模块。
- **命名空间转换**：将C++中的命名空间转为JavaScript的命名空间，并对其简化。示例代码如下：

```
1  ns_map:
2      "cocos2d::extension::": "cc."
3      "cocos2d::ui::": "ccui."
4      "cocos2d::": "cc."
5      "spine::": "sp."
6      "cocostudio::": "ccs."
7      "cocosbuilder::": "cc."
8      "CocosDenshion::": "cc."
```

- **JavaScript变量转换为C++变量**：`ok &=` 后面的函数，为转换函数。例如，`jsval_to_ccrect`函数为将JavaScript传过来的`cc.rect`变量转为C++的`Rect`。代码见conversions.yaml文件中的`to_native`模块。
- **JavaScript对象转换成C++对象**：代码见conversions.yaml文件中的to_native/object模块。
- **C++变量转换成JavaScript变量**：代码见conversions.yaml文件中的from_native模块。
- **C++对象转换成JavaScript对象**：代码见conversions.yaml文件中的from_native/object模块。

16.4　绑定自己的 C++类

有了Bindings Generator工具，绑定则不是想象中那么复杂了，我们只需写好要绑定的C++类和绑定配置文件之后，再运行Bindings Generator工具，即可完成绑定。

16.4.1　编写要绑定的 C++类

首先，编写一个简单的Person类，这个类定义了4个构造函数、一个析构函数、6个针对类属性的get和set方法以及两个简单的类行为函数，代码如下：

Person.h

```
1  #ifndef Person_h
2  #define Person_h
3  #include <string>
4  class Person{
5  // [初始化相关]
6  public:
7      Person();
8      Person(const std::string& name);
9      Person(const std::string& name, int age);
10     Person(const std::string& name, int age, float height);
11     ~Person();
12 // [属性定义]基础属性
13 protected:
14     std::string m_sName;
15     int m_nAge;
16     float m_fHeight;
17 // [方法声明]类自身行为
18 public:
19     void onGreetings();     // 打招呼
20     void onSpeakWithContent(const std::string& content);// 说话
21 // [方法声明]属性getter && setter
22 public:
23     const std::string& getName();
24     void setName(const std::string& name);
25     int getAge();
26     void setAge(int age);
27     float getHeight();
28     void setHeight(float height);
29 };
30 #endif /* Person_h */
```

Person.cpp

```
1  #include "Person.h"
2  #include <iostream>
3  using namespace std;
4  // ......
5  // 打招呼
6  void Person::onGreetings() {
7      std::cout << "Hi, I am person";
```

```
8  }
9  // 说话
10 void Person::onSpeakWithContent(const std::string& content){
11     std::cout << content;
12 }
13 // ......
```

当编写好C++文件之后，即可开始绑定。

16.4.2 生成 C++类的自动绑定代码

实际上，Cocos引擎提供给开发者生成绑定代码的方式有两种。其一，通过绑定环境的test测试工程。其二，扩展引擎中生成自动绑定代码的工具，这也是推荐的做法。我们先一起来看看如何通过test测试工程生成绑定代码。

1. 通过test测试工程绑定

test工程位于cocos2d-x/tools/bindings-generator路径下，其目录结构如图16-12所示。其中，simple_test文件夹保存需要绑定的C++类文件，test.sh和test.bat分别为Mac OS X和Windows操作系统中生成自动绑定代码的脚本工具，test.ini文件保存绑定的规则，剩下两个以sample结尾的文件保存一些环境配置信息，例如Python安装路径、NDK_ROOT、clangllvmdir路径等配置在user.cfg.sample中，如果你需要用它来配置这些信息，则应当将.sample后缀去掉，才能保证test脚本能正确读取到它，而userconf.ini则为运行test脚本后输出的配置信息。在Mac OS X中，可忽略不管这两个文件，在Windows系统中，也可通过直接修改test.bat文件的方式取代user.cfg。所以，不管如何，这两个文件都可直接忽略不管。

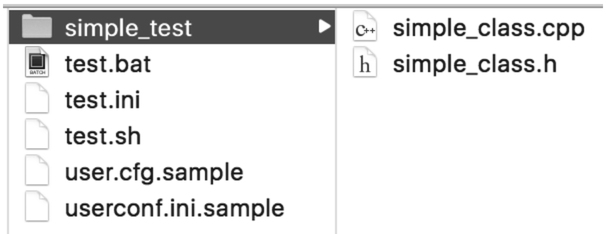


图16-12 test目录结构

了解完这些信息之后，就可以开始绑定了，绑定的步骤如下。

(1) 复制测试工程。在cocos2d-x/tools/bindings-generator路径下新建一个文件夹并对其命名，例如笔者将其命名为MyBinding，然后复制simple_test、test.bat、test.ini和test.sh到MyBinding文件夹中。

(2) 添加C++文件。删除simple_test中的simple_class.h和simple_class.cpp文件，并对simple_test进行重命名，例如person_binding，然后将需要绑定的类所在的C++文件复制到person_binding下。

(3) 编写.ini规则文件。其主要步骤有如下5个步骤，其中“-”和“+”符号为模拟Git中的代码变动，表示代码的删除和添加。具体代码见随书代码资源的Unit_16_JavaScript_Binding/ByTest/MyBinding/test.ini文件。

步骤1	说明	将prefix的值修改为auto_person_binding，自动绑定工具最后生成的文件都会以此值命名或者作为前缀，例如auto_person_binding.hpp、auto_person_binding.cpp和auto_person_binding_api.js
	-	`prefix = autogentestbindings`
	+	`prefix = auto_person_binding`
步骤2	说明	填写要绑定的类，多个类之间以空格隔开
	-	`classes = SimpleNativeClass`
	+	`classes = Person`
步骤3	说明	删除test测试工程C++头文件路径
	-	`simple_test_headers = -I%(cxxgeneratordir)s/test/simple_tess`
	-	`extra_arguments = %(android_headers)s %(clang_headers)s %(android_flags)s %(clang_flags)s %(simple_test_headers)s %(extra_flags)s`
	+	`extra_arguments = %(android_headers)s %(clang_headers)s %(android_flags)s %(clang_flags)s %(extra_flags)s`
步骤4	说明	填写要绑定的C++类头文件路径
	-	`headers = %(cxxgeneratordir)s/test/simple_test/simple_class.h`
	+	`headers = %(cxxgeneratordir)s/MyBinding/person_binding/Person.h`
步骤5	说明	填写命名空间，此步骤可选，不过建议加上
	+	`target_namespace = jf`

(4) 修改并运行生成自动绑定代码的脚本。

在Mac OS X和Windows操作系统中，生成绑定代码的脚本是不一致的，Mac OS X中采用shell脚本，而Windows中为.bat批处理脚本。

- **Mac OS X操作系统**：修改test.sh文件中最后一行`LD_LIBRARY_PATH`的值，将`${CXX_GENERATOR_ROOT}/test`改为`${CXX_GENERATOR_ROOT}/MyBinding`。然后打开终端工具，运行如下命令，即可生成绑定代码：

```
1  cd /pathTo/cocos2d-x/tools/bindings-generator/MyBinding
2  ./test.sh
```

- **Windows操作系统**：在test.bat文件中的第10行左右，打开`PYTHON_ROOT`和`NDK_ROOT`的注释，并为其设置值，示例代码如下：

```
1  set PYTHON_ROOT=C:/Python27
2  set NDK_ROOT=C:/Users/LingJianfeng/Desktop/jsb/ndk/android-ndk-r9d
```

设置好`PYTHON_ROOT`和`NDK_ROOT`的值之后，将test.bat文件中`%CXX_GENERATOR_ROOT%/test`改为`%CXX_GENERATOR_ROOT%/MyBinding`，然后打开命令行程序（cmd.exe），运行如下命令，即可生成绑定代码：

```
1  cd /pathTo/cocos2d-x/tools/bindings-generator/MyBinding
2  test.bat
```

说明 上述两个代码段中的pathTo为读者自己电脑上引擎所存放的路径，后面的pathTo也是如此，例如笔者Mac OS X的路径是：/Users/Jeff/Documents/Cocos/cocos2d-x/tools/bindings-generator/MyBinding。
Windows上的路径是：C:\Users\LingJianfeng\Desktop\cocos2d-x\tools\bindings-generator\MyBinding。

运行生成绑定代码的脚本之后，在当前文件路径下生成一个simple_test_bindings文件夹，其中包含一个api文件夹，以及一个.hpp和.cpp的文件，如图16-13所示。

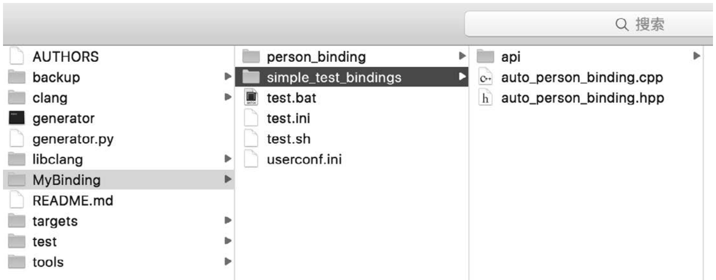


图16-13 绑定代码生成的结果

虽然此时生成了绑定代码，但是值得注意的是，Person类并没有继承cocos2d::Ref对象，也就是说通过Person类创建的实例对象的内存需要我们自己管理，这和Cocos引擎的设计理念显得有些出入。若将Person继承自cocos2d::Ref，则需在Person.h中引入cocos2d.h头文件，但是此时再运行脚本绑定工具，则会报找不到cocos2d.h文件的错误，这是因为test绑定测试工程中并没有把Cocos2d-x引擎相关的类绑定进来（见16.2.1节）。所以，只要我们需要绑定的类和Cocos2d-x引擎有交集，就可将此绑定行为归纳为对引擎的扩展，应当考虑通过引擎绑定代码生成的genbindings.py脚本生成绑定代码，这样会简单和方便许多。

2. 通过引擎绑定代码生成的genbindings.py脚本

genbindings.py文件位于cocos2d-x/tools/tojs文件夹下，它是生成引擎绑定代码的脚本，我们可以借助它来生成需要的绑定代码。但是在这之前，我们先创建一个Cocos2d-JS游戏工程HelloJSB。接着，我们来看下如何使用genbindings.py脚本，其步骤如下。

(1) 制作genbindings.py的副本文件。在HelloJSB/frameworks/cocos2d-x/tools/tojs文件夹下新建一个Python脚本并对其命名，例如我将它命名为jeff_genbindings.py，然后将genbindings.py中所有的代码复制到jeff_genbindings.py里面，再将cmd_args的值改为如下代码：

```
1  cmd_args = {'person.ini':('custom-model','jsb_jeff_external_auto'),}
```

其中，person.ini表示等下需要编写的配置文件，custom-model为模块名称，它对应到person.ini文件中的第一行代码，而jsb_jeff_external_auto则为最终生成绑定代码的文件名称，生成的绑定代码

文件将默认被存放在cocos2d-x/cocos/scripting/js-bindings/auto文件夹下。当然，你也可以修改生成代码的存放路径，例如我将它放在HelloJSB/frameworks/runtime-src/Classes/jeff/auto文件下，那么只需修改jeff_genbindings.py文件中output_dir变量的值即可，代码如下：

```
1  output_dir = '%s/../runtime-src/Classes/jeff/auto' % project_root
```

(2) 添加C++文件。把之前写好的Person.h和Person.cpp文件复制到HelloJSB/frameworks/runtime-src/Classes/jeff下。另外，值得说明的是，我并不推荐你将自己编写的C++类以及自动绑定生成的代码集成到cocos2d-x引擎当中，应当将这些放在项目的frameworks/runtime-src/Classes目录下，这是一种正确的做法，也是官方推荐的做法。

此时，可将我们之前编写好的`Person`类做一个修改，让它继承自`cocos2d::Ref`，从而支持Cocos2d-x引擎的内存自动管理机制，并将Log的输出方式也改为`CCLOG`宏。修改后的代码如下：

修改后的Person.h

```
1  // ......
2  #include "cocos2d.h"
3  class Person : public cocos2d::Ref{
4  // [初始化相关]
5  public:
6      Person();
7      Person(const std::string& name);
8  // ......
```

修改后的Person.cpp

```
1  // ......
2  // 打招呼
3  void Person::onGreetings() {
4      CCLOG("Hi, I am person");
5  }
6  // 说话
7  void Person::onSpeakWithContent(const std::string& content){
8      CCLOG("%s", content.c_str());
9  }
10 // ......
```

(3) 编写.ini配置文件。

在<(1) 制作genbindings.py的副本文件>中，我们指定了配置文件的名称为person.ini，所以这里我们可以通过复制出一个cocos2dx.ini副本文件，然后将其命名为person.ini，然后再对person.ini编辑即可，编辑的步骤有如下几个步骤。

步骤1	说明	更改模块名称，此参数对应jeff_genbindings.py中的`cmd_args`字典
	-	`[cocos2d-x]`
	+	`[custom-model]`
步骤2	说明	更改前缀信息，但是此值在jeff_genbindings.py中的`cmd_args`参数强制修改了，所以此值此处无效了，可忽略
	-	`prefix = cocos2dx`
	+	`prefix = user_external`

（续）

步骤3	说明	添加命名空间，此ini文件中绑定的所有类都将采用这个命名空间。在JavaScript中，将被创建成类似`'ns = ns = || {}'`格式的代码
	-	`target_namespace = cc`
	+	`target_namespace = jf`
步骤4	说明	指定需要解析的头文件，它会递归解析
	-	`headers = %(cocosdir)s/cocos/cocos2d.h ......`
	+	`headers = %(cocosdir)s/../runtime-src/Classes/jeff/Person.h`
步骤5	说明	填写要绑定的类，用空格隔开，支持正则表达式
	-	`classes = New.* Sprite SpriteBatchNode ......`
	+	`classes = Person`
步骤6	说明	指定需要继承的类
	-	`classes_need_extend = Node __NodeRGBA Layer.* ......`
	+	`classes_need_extend = Person`
步骤7	说明	将skip、rename_functions 、rename_classes等值置空
	+	`skip =`
	+	`rename_functions =`
	+	`rename_classes =`
	+	`classes_have_no_parents =`
	+	`abstract_classes =`

(4) 复制Bindings Generator工具。将cocos2d-x/tools/bindings-generator文件夹复制到HelloJSB/frameworks/cocos2d-x/tools目录下。

(5) 运行生成自动绑定代码的脚本。

当完成如上步骤之后，一切处于就绪状态，只需运行生成自动绑定代码的脚本即可，操作如下。

在Mac OS X操作系统中，打开终端，运行如下命令：

```
1  cd /Users/Jeff/Desktop/HelloJSB/frameworks/cocos2d-x/tools/tojs
2  python ./jeff_genbindings.py
```

在Windows操作系统下，通过命令行程序运行如下命令：

```
1  cd C:\Users\LingJianfeng\Desktop\HelloJSB\frameworks\cocos2d-x\tools\tojs
2  python jeff_genbindings.py
```

运行脚本之后，会在HelloJSB/frameworks/runtime-src/Classes/jeff/auto目录下生成一个C++的.hpp和.cpp文件，以及在auto/api下生成一个以.js为后缀的api文档文件，如图16-14所示。

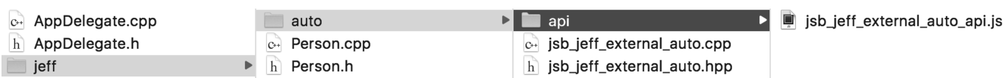


图16-14　genbindings.py生成的绑定代码

3. 将绑定生成的代码集成到项目工程中

接下来，我们需要针对不同的平台进行集成操作，可分为如下两个分支。

(1) iOS和Mac OS X的集成

在Mac OS X，你的电脑上必须装有Xcode开发软件，然后通过下面几个步骤，便可集成成功。

1) **添加绑定的类到项目工程**。在安装好Xcode的情况下，双击打开HelloJSB/frameworks/runtime-src/proj.ios_mac/HelloJSB.xcodeproj工程文件，然后将jeff文件夹拖曳到项目工程中，勾选Create groups、HelloJSB-mobile、HelloJSB-desktop按钮，如图16-15所示。添加Person文件到工程后，Xcode的目录结构如图16-16所示。

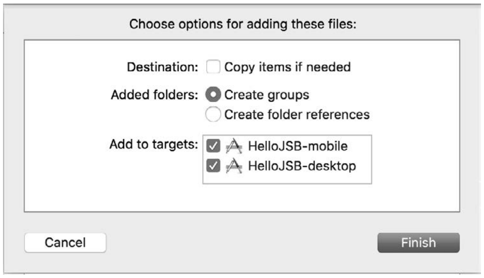


图16-15　将jeff文件夹添加到HelloJSB工程中

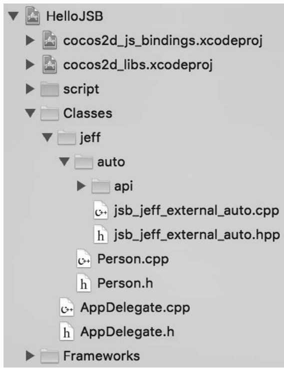


图16-16　添加Person文件到工程后，Xcode的目录结构

2) **注册绑定**。当头文件添加完毕后，便可进行函数注册。打开AppDelegate.cpp文件，引入jsb_jeff_external_auto.cpp文件，并在`AppDelegate::applicationDidFinishLaunching`方法中对绑定函数进行注册，代码如下：

AppDelegate.cpp

```
1  #include "AppDelegate.h"
2  #include "jeff/auto/jsb_jeff_external_auto.hpp"// 引入绑定代码文件
3  // ......
4  bool AppDelegate::applicationDidFinishLaunching(){
5      // ......
6      ScriptingCore* sc = ScriptingCore::getInstance();
7      sc->addRegisterCallback(register_all_cocos2dx);
8      sc->addRegisterCallback(register_cocos2dx_js_core);
9      sc->addRegisterCallback(jsb_register_system);
10     // 自定义绑定函数注册
11     sc->addRegisterCallback(register_all_user_external);
12     // ......
13 }
```

3) **测试绑定结果**。此时，便完成了C++的`Person`类到JavsScript的绑定了。我在main.js中的`cc.game.onStart`函数中键入如下代码来测试：

```
1  var man = new jf.Person("凌建风", 23, 168);
2  man.onGreetings();  // 打招呼
3  cc.log("我的名字是: ", man.getName());
4  cc.log("我的年龄是: ", man.getAge());
5  cc.log("我的身高是: ", man.getHeight());
```

运行之后，在控制台看到的输出结果如图16-17所示。到此，iOS和Mac OS X平台下的绑定成功。



图16-17　iOS和Mac OS X集成测试的结果

(2) Android的集成

Android的集成，需要在你的电脑上安装Eclipse或者Android Studio。这里我以Eclipse为例，演示一下如何将绑定的代码集成到Android中。

1) Android.mk文件修改。打开HelloJSB/frameworks/runtime-src/proj.android/jni文件夹下的Android.mk文件，添加相关cpp文件以及cpp文件所在的路径，这一步只需修改`LOCAL_SRC_FILES`和`LOCAL_C_INCLUDES`的值即可，代码如下：

```
1  // ......
2  LOCAL_SRC_FILES := hellojavascript/main.cpp \
3                  ../../Classes/jeff/Person.cpp \
4                  ../../Classes/jeff/auto/jsb_jeff_external_au to.cpp \
5                  ../../Classes/AppDelegate.cpp
6
7  LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../Classes \
8                   $(LOCAL_PATH)/../../Classes/jeff \
9                   $(LOCAL_PATH)/../../Classes/jeff/auto
10 // ......
```

2) 编译Android项目。把HelloJSB导入到Eclipse中之前，要先编译下HelloJSB工程。例如，在Mac OS X下，打开终端，运行如下指令即可，这在Windows下也是一样的：

```
1  cd /Users/Jeff/Desktop/HelloJSB
2  cocos compile -p android
```

编译成功之后，终端显示的结果如图16-18所示。



图16-18　Android工程编译成功

3) 测试绑定结果。此时可将HelloJSB导入到Eclipse当中（见2.5节），运行项目，Eclipse控制台的输出结果如图16-19所示。



图16-19　Android集成测试结果

16.5　实例——绑定 SQLite3

在第8章中，我曾提到过Cocos2d-JS本身没有将SQLite绑定到JavaScript上，需要通过代码绑定的方式自行实现。那么，在本章实例中，我们就来绑定下SQLite3数据库。

16.5.1　下载 SQLite3 源代码

首先，你需要去www.sqlite.org/download.html下载SQLite3源码，即sqlite3.h和sqlite3.c两个文件，然后还需要一个包装类，这个包装类基于wrapper（下载地址为www.adp-gmbh.cn/sqlite/wrapper.html）做了一些修改，主要是加入了平台初始化信息以及Android目录权限问题，具体可见随书代码资源中的HelloSqlite/frameworks/runtime-src/Classes/sqlite/ SQLiteWrapper.cpp文件。

16.5.2　创建 HelloSqlite 工程

下载所需的相关文件之后，便可以创建一个Cocos2d-JS工程，我将其命名为HelloSqlite，然后将SQLite3相关的文件放在HelloSqlite/frameworks/runtime-src/Classes/sqlite文件夹下，如图16-20所示。



图16-20　SQLite3相关的源文件

16.5.3　生成绑定代码

接下来，需要通过genbindings.py脚本生成SQLite3的绑定代码，此步骤可参考16.4.2节，最终生成的绑定代码依旧选择放在HelloSqlite/frameworks/runtime-src/Classes/sqlite/auto文件夹下。

16.5.4 注册绑定

打开AppDelegate.cpp文件，引入jsb_sqlite_external_auto.hpp文件，并在AppDelegate::applicationDidFinishLaunching方法中对绑定函数进行注册，代码如下：

AppDelegate.cpp

```
1  #include "AppDelegate.h"
2  // 引入自动绑定代码文件
3  #include "sqlite/auto/jsb_sqlite_external_auto.hpp"
4  // ......
5  bool AppDelegate::applicationDidFinishLaunching(){
6      // ......
7      ScriptingCore* sc = ScriptingCore::getInstance();
8      sc->addRegisterCallback(register_all_cocos2dx);
9      sc->addRegisterCallback(register_cocos2dx_js_core);
10     sc->addRegisterCallback(jsb_register_system);
11     // 绑定函数注册
12     sc->addRegisterCallback(register_all_sqlite_external);
13     // ......
14 }
```

16.5.5 集成到平台中

在iOS和Mac OS X上，你只需打开HelloSqlite/frameworks/runtime-src/proj.ios_mac/HelloSqlite.xcodeproj工程文件，然后在Xcode中将HelloSqlite/frameworks/runtime-src/Classes/sqlite文件夹拖曳到HelloSqlite项目中即可，如图16-21所示。



图16-21 添加sqlite文件到工程后Xcode的目录结构

Android的集成需要修改HelloSqlite/frameworks/runtime-src/proj.android/jni/Android.mk文件，实际上只是添加相关需要编译的文件以及文件所在的路径，代码如下：

```
1  // ......
2  LOCAL_SRC_FILES := hellojavascript/main.cpp \
3                     ../../Classes/sqlite/sqlite3.c \
4                     ../../Classes/sqlite/SQLiteWrapper.cpp \
5                     ../../Classes/sqlite/auto/jsb_sqlite_external_auto.cpp \
6                     ../../Classes/AppDelegate.cpp
7  LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../Classes \
8                      $(LOCAL_PATH)/../../Classes/sqlite \
9                      $(LOCAL_PATH)/../../Classes/sqlite/auto
10 // ......
```

16.5.6　测试结果

在前面的步骤中，我们已经完成了SQLite3数据库的绑定。接下来，需要编写一些代码，来测试下之前绑定的数据库能否正常工作。

首先，需要自己编辑一个数据库.db文件，或者你也可以将随书代码资源中编辑好的data.db文件直接复制过来，它位于随书代码资源中的HelloSqlite/res文件夹下。然后通过如下代码从data.db数据库文件查询出表的内容：

```
1  loadSqlite : function(sender){
2      if (!cc.sys.isNative) { return; }
3      this.database = new sql.SQLiteWrapper();
4      this.databasePath = this.database.initializing("data.db","res","");
5      // 打开[数据库]
6      if (this.database.open(this.databasePath)){
7          cc.log("数据库打开[成功]! ");
8          var st = this.database.statement("select * from userInfo");
9          var userDataArray = [];
10         while(st.nextRow()){
11             var userData = new UserData();
12             userData.uid = parseInt(st.valueString(0));
13             userData.name = st.valueString(1);
14             userData.level = st.valueString(2);
15             userData.desc = st.valueString(3);
16             userData.gold = parseInt(st.valueString(4));
17             userData.diamond = parseInt(st.valueString(5));
18             userDataArray.push(userData);
19         }
20         // 数据遍历
21         for(var data in userDataArray){
22             cc.log("userData:" + userDataArray[data].toString());
23         }
24     }else{
25         cc.log("数据库打开[失败]! ");
26     }
27 }
```

其中，第11行代码所创建的对象实际上就是一个单纯的数据对象类，其代码如下：

```
1  var UserData = cc.Class.extend({
```

```
2        uid      : 0,
3        name     : "",
4        level    : 0,
5        desc     :"",
6        gold     :0,
7        diamond :0,
8        toString:function(){
9            return this.name + "[" + this.uid + "] 等级: " + this.level + " 描述: " +
             this.desc + " 金币: " + this.gold + " 钻石: " + this.diamond;
10       }
11 });
```

运行代码，在Xcode和Eclipse中控制台的输出日志如图16-22和图16-23所示。



图16-22　SQLite测试Xcode控制台的输出结果



图16-23　SQLite测试Eclipse控制台的输出结果

16.6　小结

通过本章的学习，我们知道了Cocos2d-JS可分为4层，最底层为C++编写的Cocos2d-x游戏引擎，往上一层是JavaScript脚本引擎。Cocos2d-JS采用的脚本引擎是Mozilla的SpiderMonkey引擎，SpiderMonkey是世界上第一个JavaScript引擎，也是目前JavaScript执行性能最优秀的引擎之一。再往上一层，是脚本绑定层，俗称胶水层，胶水层代码使得C++和JavaScript两种语言可以相互调

用。最后，最上面的那层，便是脚本游戏层，这一层都是游戏的业务逻辑处理代码。另外，我们还学习了自动绑定技术，深入探究绑定技术背后的原理，并在最后的章节实例中完成了SQLite3数据库的绑定。

16.7　参考资源

本章的参考资源如下。

- Travis CI：https://travis-ci.org/。
- SpiderMonkey引擎：https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey。
- Bindings Generator工具：https://github.com/cocos2d/bindings-generator/。
- Android.mk文件语法规范：http://blog.csdn.net/smfwuxiao/article/details/8530742。
- cvp panda的博客：http://cvp.cocos.com/blog/265。
- cvp panda的博客：http://cvp.cocos.com/blog/266。
- Cocos2d-JS 3.0 RC3中通过JSB 使用SQLite3： http://www.cocoachina.com/bbs/read.php?tid=226949&page=1&toread=1#tpc。

第四部分

实 战 篇

第 17 章

《保卫萝卜2》实战

正所谓“台上一分钟，台下十年功”，我们经过了前面的学习，已经系统完整地掌握了Cocos2d-JS游戏引擎，此时此刻，你可以抱着“是骡子是马牵出来遛遛”的心态来学习本书的最后一章。但是，我相信当你阅读到本章时，你已然是一匹可以在草原中策马奔腾的骏马了。那么，来吧，怀着“I Can I Up”的心态和我一起完成《保卫萝卜2》项目的开发。

本章内容：

- 关卡选择场景开发
- 游戏管理对象`GameManager`的开发
- 游戏玩法场景开发

17.1 关卡选择场景开发

在10.12节中，我们初步开发了关卡选择场景，完成了滚动背景的开发。那么在本节中，我们将一起完成关卡选择场景的开发，完成之后的效果如图17-1所示。



图17-1　关卡选择场景

对图17-1进行分析可以得出，关卡选择场景有一个背景滚动视图（ScrollView），所有的节点都添加在此视图中，例如第10章添加进去的14张背景图片，还有本节中将要开发的关卡连线、关卡选择按钮以及关卡按钮类似涟漪的特效等。

在图17-1中可以看到，滚动视图中的背景图片上，美术已经标记好了每个关卡按钮的位置，即数字1、2、3等。但是，这仅仅只是在背景图片中标记出来，程序并不能直接获取出位置，那么我们应当如何获取这些位置呢？我想，你应该想到瓦片地图中的对象层，通过瓦片地图编辑器，我们可以很方便快速地把这些位置标记出来。下面就来看下如何在瓦片地图中标记这些位置。

17.1.1 使用瓦片地图编辑器标记关卡按钮坐标

首先，在Tiled Map Editor编辑器中新建一个瓦片地图，并对其命名，例如TiledMap.tmx，然后把这个文件保存在res/ChooseLevel/Map文件夹下。设定地图大小为206 × 10块，瓦片大小为64 × 64，这是因为每一张背景图片的大小是943 × 640，总的有14张，所有背景加起来的大小是宽为943 × 14 = 13202，高为640，每个图块的大小设为64 × 64，那么，整张瓦片地图的大小大约是宽为13202 / 64 = 206.28125， 高为640 / 64 = 10 。又因为地图大小的取值为整数，所以对206.28125进行向下取整，得到206。你可能会问，向下取整后有其他影响吗？毫无疑问，没有什么影响。你也可以选择向上取整，只要保证14张背景图都能在地图编辑器中完全显示出来即可。因为我们只是需要借用这些背景图片来作为参考，从而标记出关卡选择按钮的坐标位置。

接下来，在Tiled Map Editor编辑器中的操作步骤如下。

(1) 将14张背景图片一一添加到地图编辑器的“图块”面板中。

(2) 对图层中的默认图层进行重命名，例如bg。

(3) 选中“图章刷”工具。

(4) 选中“图块”面板中所有的图块。

(5) 在地图编辑区“印出”背景图片。

(6) 重复第(4)步和第(5)步，完成剩下13张背景图的编辑。

(7) 新建一个对象层，并对其命名，例如`point`。

(8) 选中`point`对象层，然后在菜单栏中选择“插入矩形”工具，然后在地图编辑区中根据背景图片中的位置标记，在对应的位置插入矩形，重复此步骤，直到完成135个标记。

完成之后，效果图如图17-2所示。

说明 在“图块”面板中，选中第一块图块之后，按住Shift键，再点击最后一块图块，便选择了所有的图块。

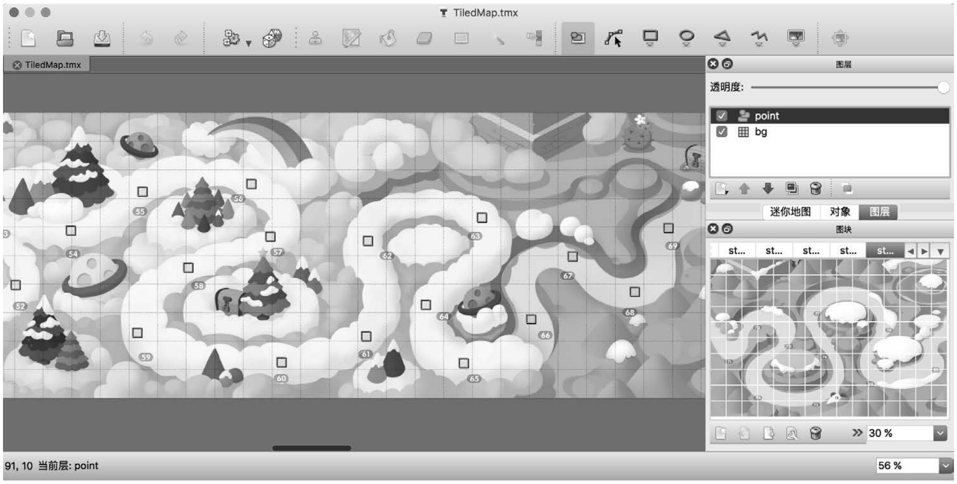


图17-2　标记关卡按钮的位置

如此，便完成了135关关卡选择按钮的位置标记，然后只要在程序中获取point对象层中的对象，再遍历处理即可。

现在，回到代码中。我给CLBackgroundLayer对象（位于src/Scene/ChooseLevel/Layer/Background.js）声明了两个属性zOrderMap和routeButtonArray，然后编写了一个loadProperty()函数，用来配置这些属性的值，相关代码如下：

```
1  var CLBackgroundLayer = cc.Layer.extend({
2      scorllView          : null,          // 滚动视图
3      zOrderMap           : {},            // 层级枚举
4      routeButtonArray    : [],            // 关卡按钮数组
5      onEnter : function () {
6          this._super();
7          // 加载[属性]
8          this.loadProperty();
9          //......
10     },
11     // 加载[属性]
12     loadProperty : function(){
13         this.zOrderMap.route = 1;                // [层级]道路
14         this.zOrderMap.routeButtonEffect = 5;    // [层级]按钮特效
15         this.zOrderMap.levelButton = 10;         // [层级]按钮
16
17         this.routeButtonArray = [];              // 清空按钮数组
18     }
19 });
```

其中，zOrderMap对象用来模拟枚举，其中保存着各类节点的层级，这样使得各类节点的层级便于管理。routeButtonArray是一个数组，用来保存后面将要创建的关卡按钮。

接下来，可以读取瓦片地图中的配置信息，然后创建出关卡选择按钮，代码如下：

```
1  // 加载[瓦片地图]
2  loadTiledMap : function(){
3      var node = new cc.TMXTiledMap("res/ChooseLevel/Map/TiledMap.tmx");
4      var objectGroup = node.getObjectGroup("point");
5      var objs = objectGroup.getObjects();
6      for (var i = 0; i < objs.length; i++) {
7          var button = new ccui.Button();
8          this.scorllView.addChild(button, this.zOrderMap.levelButton, i);
9          this.routeButtonArray.push(button);
10         // 图片纹理
11         var texture = "res/ChooseLevel/stagepoint_adv.png";
12         // 编辑器中配置的属性
13         if (objs[i].isBoos == "YES") {
14             texture = "res/ChooseLevel/stagepoint_boss.png";
15         }else if (objs[i].isTime == "YES") {
16             texture = "res/ChooseLevel/stagepoint_time.png";
17         }else if (objs[i].isChange == "YES"){
18             texture = "res/ChooseLevel/stagepoint_chance.png";
19         }else{
20             texture = "res/ChooseLevel/stagepoint_adv.png";
21         }
22         button.loadTextures(texture, texture, "");
23         button.setPosition(objs[i].x, objs[i].y);
24         button.setTag(i);
25         button.addTouchEventListener(this.onLevelButtonEvent, this);
26     }
27 }
```

在第4行中，我获取了瓦片地图中的point对象层。但是这样做是不够的，还应该把point对象层中所有的对象获取出来，第5行就是做了这样的一件事情。当获取所有的对象信息后，就可以遍历它们了，然后根据它们的信息创建并配置按钮，这些配置信息主要是位置坐标、是否是Boss关卡（见第13行）、是否是时间关卡（见第15行）、是否是随机关卡（见第17行）等。这些配置也是在瓦片地图编辑器中配置的，它实际上是point对象层中某个对象的自定义属性值，如图17-3所示。



图17-3　关卡类型配置

第24行是一行不起眼但又非常重要的代码，它设定了当前按钮的tag值，实际上就是当前的关卡数，索引从0开始，即0表示第1关，1表示第二关，以此类推。

最后，第25行给关卡按钮添加了触摸事件监听器。事件响应函数onLevelButtonEvent()的逻辑代码如下：

```
1  // 事件[关卡按钮]
2  onLevelButtonEvent : function(sender, type){
3      switch (type) {
4          case ccui.Widget.TOUCH_ENDED:
5              var level = sender.getTag();      // 当前等级
6              // TODO：加载关卡数据，进入游戏
7          break;
8      }
9  }
```

很显然，在onLevelButtonEvent()函数中，我们要做的事情就是将当前按钮的tag值获取出来，这个tag值就是关卡，然后根据tag值加载对应关卡的数据，最后进入游戏场景。但是，不要急，本节先完成关卡选择页面的开发，关卡数据加载等将在后面讲解，这里可以留下一个TODO标记。

最后，在CLBackgroundLayer.onEnter()方法中调用loadTiledMap()函数，运行一下项目，得到的结果如图17-4所示。至此，135个关卡按钮开发完成。

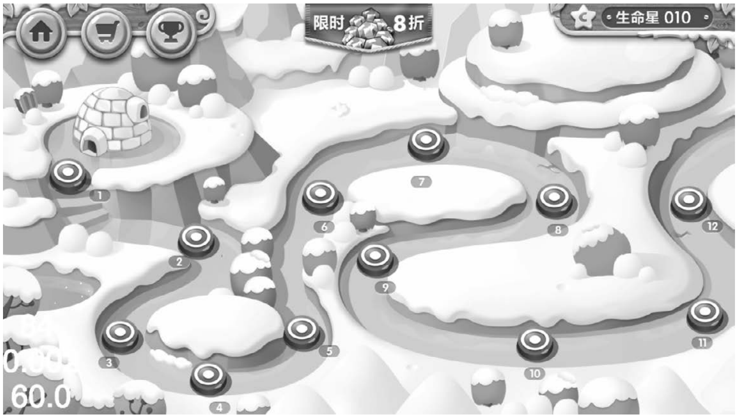


图17-4　完成的关卡按钮

17.1.2　关卡之间的连线开发

从图17-1中可以看到，《保卫萝卜2》在完成某一个关卡之后，会有一条“走过的路”，这个功能的实现代码比较简单，具体如下：

```
1  // 加载[关卡道路]
2  loadRoute: function (level) {
```

```
3       var node = null;
4       for (var i = 0; i < level - 1; i++) {
5           node = new cc.Sprite("#route_" + (i + 1) + ".png");
6           if (i % 10 == 9 ) {
7               node.setAnchorPoint(0, 0.5);
8           }
9           node.x = node.width / 2 + Math.floor(i / 10) * node.width;
10          node.y = this.scorllView.getInnerContainerSize().height / 2;
11          this.scorllView.addChild(node, this.zOrderMap.route);
12      }
13 }
```

loadRoute(level)函数接收一个Number类型的参数，这个参数表示关卡，整个loadRoute(level)函数的代码实现不过12行而已。或许你会很诧异，就这短短的12行代码就实现了道路的功能？实际上，道路的实现代码可以如此简单，是因为在图片资源上，已经默认帮我们对好坐标位置了。但是需要注意的是，每隔10张需要调整当前道路的锚点，这也是受图片资源的影响。浏览一下随书资源res/ChooseLevel/Route文件夹下的道路图片，便可非常直观地感受到这一设计技巧。

另外，第5行取的图片是精灵表单中的资源，所以你应该可以想到我会在ChooseLevelScene（位于src/Scene/ChooseLevel/ChooseLevel.js）对象中的loadResource()函数加载了精灵表单资源。那么，这里到底为什么要用精灵表单呢？这是因为每一张道路图片的大小都是960 × 640，但是每张图片中的“有效内容”又是非常少，大部分都是透明区域，如果把这些道路图片做成精灵表单，那么134张的图片就“压缩”到了如图17-5所示的精灵表单，这样的话，如果你是在Web环境下开发《保卫萝卜2》，那么在没有使用精灵表单之前，需要从服务器下载134张道路图片，但是用了精灵表单之后，只需下载一个.plist文件和一张大小为2048 × 2048的精灵表单即可。这不管是在Native上，还是在Web中，都将极大地提高了资源读取的速度。

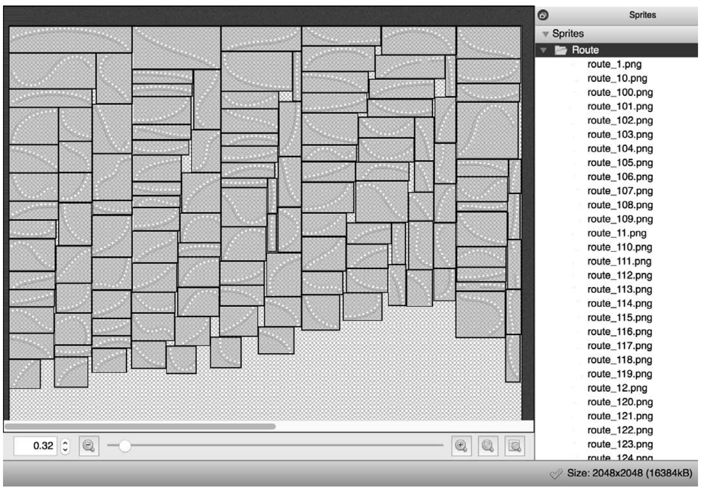


图17-5 制作道路精灵表单

说明　在实际开发中，做Demo阶段，一般不会把所有的图片都制作成精灵表单，而是先用单张的图片，即小图，或称碎图。当项目开发进入到后期时，例如打包测试，才将所有的图片制作成精灵表单。

最后，在`CLBackgroundLayer`的`onEnter()`方法中调用`loadRoute(level)`函数，传递一个关卡数，例如我传递了11，那么再运行项目，可以看到的效果如图17-6所示。



图17-6　道路开发完成

17.1.3　关卡按钮特效节点开发

关卡按钮特效的实现，实际上是在一个节点中创建了3个精灵，然后特效节点依次重复运行一组组合动作，便产生了"涟漪"的波纹效果，具体参见src/Scene/ChooseLevel/Node/RouteButton-Effect.js文件。

17.2　游戏管理对象 `GameManager` 的开发

塔防类游戏一般都有许多的关卡，每个关卡的玩法都一样，不同的只是在关卡数据配置上。所以，我可以设计一个游戏管理对象（`GameManager`），用来管理每一关所需要的数据。

在本书编写的Carrot项目中，我将数据分为两大块，一块为关卡数值数据，另外一块为关卡道路、障碍物配置等数据。这两块数据在实际开发中，一般都是由策划去配置的。前者我用代码直接编写，后者使用Tiled Map Editor编辑器编辑。整个游戏核心玩法的结构图如图17-7所示。

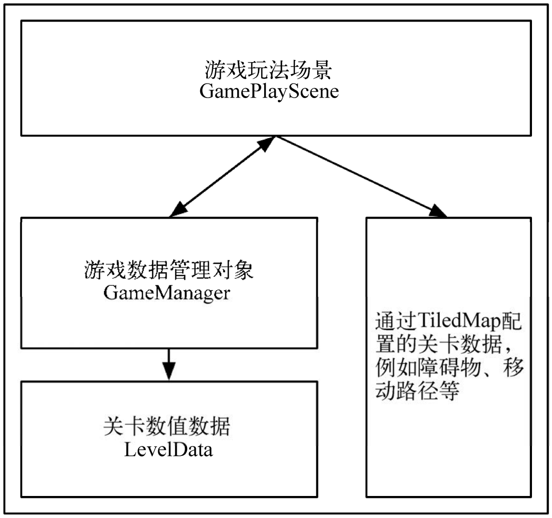


图17-7　游戏核心玩法设计结构

17.2.1　关卡数值配置

LevelData实际上为一个数组，其中保存着每一关的关卡数值数据，例如第1关的数值如下（详见src/Data/LevelData.js）：

```
1  var LevelData = [
2      {   // 第1关
3          themeID         : 1,            // 主题
4          group           : 6,            // 组数
5          gold            : 800,          // 初始金币
6          enemyInterval   : 1,            // 刷怪时间间隔
7          groupInterval   : 1,            // 组数时间间隔
8          levelName       : "level 1",    // 关卡名字
9          blood           : 10,           // 萝卜血量
10
11         monsterGroup : [     // 每一关的怪物数据
12             {   // 第1组
13                 index : 1,
14                 team : [
15                     { name : "L11", count : 5, blood : 5.0, speed : 180 }
16                 ]
17             }
18             // ......
19         ]
20     }
21 ];
```

可以看到，每一关都配置有主题、组数、初始金币、刷怪时间间隔、组数时间间隔、关卡名字以及萝卜血量这些基础信息。

此外，每一关的所有怪物的基本数据都被配置在monsterGroup数组中，然后team数组又存放着当前组中每一队的怪物数据。例如，下面的配置表示的是当前组有3队怪物，第1队怪物的名字是L31，数量为5，血量为40，怪物移动速度为180，第2队和第3队同理：

```
1  team : [
2      { name : "L31", count : 5, blood : 40.0, speed : 180 },
3      { name : "L11", count : 5, blood : 40.0, speed : 180 },
4      { name : "L21", count : 5, blood : 45.0, speed : 180 }
5  ]
```

需要注意的是，这里我让怪物的名字和怪物的图片名字对应，这样便可以通过怪物名字直接得到对应怪物的图片资源，从而创建出怪物精灵。

17.2.2　GameManager 编写

有了关卡数值数据之后，便可以开始编写游戏管理对象GameManager了。在编写GameManager对象之前，我们可以先列下需求点，即希望GameManager实现哪些功能，提供哪些接口等。

前面我已经提到过，GameManager的主要功能就是管理游戏的数据对象，《保卫萝卜2》的出怪规则是一组怪物一组怪物的出来，所以，目前你应该至少想到，GameManager至少需要提供一个读取关卡数据的接口和一个pop（弹出）下一组怪物数据的接口。

接下来，就来正式编写GameManager对象。首先，在src文件夹下新建一个Manager文件夹，然后再新建一个JS脚本文件，并将其命名为GameManager，最后在GameManager.js中定义出游戏管理对象GameManager，并在GameManager对象中声明相关的属性，代码如下：

```
1  // 游戏管理对象
2  var GameManager = {
3      level              : 0,        // 关卡[从0开始]
4      levelData          : [],       // 关卡[数据]
5      themeID            : 0,        // 主题
6      monsterGroup       : [],       // 怪物[数据]池
7      group              : 0,        // 组别
8      maxGroup           : 0,        // 组别[最大值]
9      _groupIndex        : 0,        // 组别索引[仅在遍历时使用]
10     carrotBlood        : 0,        // 萝卜的血量
11     gold               : 0,        // 初始金币
12     enemyInterval      : 0,        // 刷怪时间间隔
13     groupInterval      : 0,        // 组数时间间隔
14     levelName          : 0,        // 关卡名字
15
16     // [获取下一个怪物数据]相关属性
17     _teamIndex             : 0,        // 队伍游标
18     _teamCount             : 0,        // 队伍总数
19     _teamMonsterCount      : 0,        // 当前队伍怪物总数
20     _teamMonsterIndex      : 0,        // 当前队伍怪物游标
21     isMonsterGetFinish     : false,    // 所有怪物是否获取完毕
22
```

```
23      // [弹出下一组怪物数据]相关属性
24      _monsterDataArray    :[],     // 怪物数据二维数组
25
26      currMonsterDataPool     : [],    // [当前]怪物数据池
27      currMonsterPool         : [],    // [当前]怪物节点池
28      currBulletPool          : [],    // [当前]子弹节点池
29
30      isWin           : false,    // 是否赢了
31      // ......
32 }
```

需要说明的是，_groupIndex属性仅在GameManager对象内部遍历出所有的怪物数据时用到，它并不表示当前的组数。

第17行到第21行主要服务于遍历出当前组所有怪物数据的相关函数。

第26行到第28行保存着当前组的怪物数据、怪物节点以及子弹节点，这些在后面的代码中都将用到。

说明 在GameManager对象内，所有以下划线（_）开头的成员属性或方法都表示私有的，它们仅在GameManager对象内部使用。

1. 加载关卡数据

接下来，给GameManager对象编写一个loadLevelData()函数，用来加载指定关卡的关卡数据。相关代码如下：

```
1  // 加载[关卡数据]
2  loadLevelData : function (level) {
3      this.level          = level;
4      this.levelData      = LevelData[level];
5      this.themeID        = this.levelData.themeID;
6      this.monsterGroup   = this.levelData.monsterGroup;
7      this.group          = 0;
8      this.maxGroup       = this.monsterGroup.length - 1;
9      this._groupIndex    = 0;
10     this.carrotBlood    = this.levelData.blood;
11     this.gold           = this.levelData.gold;
12     this.enemyInterval  = this.levelData.enemyInterval;
13     this.groupInterval  = this.levelData.groupInterval;
14     this.levelName      = this.levelData.levelName;
15
16     this._teamIndex       = 0;
17     this._teamCount       = this.monsterGroup[0].team.length - 1;
18     this._teamMonsterIndex  = 0;
19     this._teamMonsterCount  = this.monsterGroup[0].team[0].count - 1;
20     this.isMonsterGetFinish = false;
21
22     this._monsterDataArray   = [];
23
```

```
24      this.currMonsterDataPool = [];
25      this.currMonsterPool     = [];
26      this.currBulletPool      = [];
27
28      this.isWin = false;
29
30      // 加载[怪物数据]
31      this._loadMonsterData();
32  }
```

loadLevelData(level)内部实现的代码就是根据传递进来的关卡编号加载对应的数据。另外需要特别注意的是，在loadLevelData(level)函数中，除了加载数据之外，还要重置对应的数组，例如第22行到第26行。这样保证了在读取当前关卡数据以及进入到下一关的时候，能保证所有的数据都干净、独立。

2. 加载所有怪物数据

在loadLevelData(level)函数中可以看到，第31行调用了_loadMonsterData()函数，此函数读取当前关卡所有的怪物数据，并保存在_monsterDataArray二维数组中，其代码如下：

```
1   // 加载[怪物数据]
2   _loadMonsterData : function(){
3       var group;  // 组
4       var team;   // 队
5       var unit;   // 只
6       var data = {};
7       this._monsterDataArray = [];
8       for (group = 0; group < this.monsterGroup.length; group++) {
9           this._monsterDataArray[group] = [];
10          for (team = 0; team < this.monsterGroup[group].team.length; team++) {
11              for (unit = 0; unit < this.monsterGroup[group].team[team].count;
                unit++) {
12                  data = this._getNextMonsterData();
13                  this._monsterDataArray[group].push(data);
14              }
15          }
16      }
17  }
```

上面的代码经过3层遍历，把所有的怪物数据都获取出来，并保存在_monsterDataArray二维数组中。举个例子，从第1组开始，数据代码如下：

```
1   {   // 第1组
2       index : 1,
3       team : [
4           { name : "L11", count : 5, blood : 5.0, speed : 180 }
5       ]
6   }
```

上面代码表示第1组有5个一样的怪物，怪物的名字（这里我以怪物图片命名，实际上你也可以理解为类型）为L11，血量为5.0，移动速度为180，那么经过_loadMonsterData()函数处理之后，第1组怪物的数据_monsterDataArray[0]的长度就是5，里面保存了第1组中所有的5个

怪物的数据。再看看第3组怪物的数据，如下：

```
1  {// 第3组
2      index : 3,
3      team : [
4          { name : "L21", count : 5, blood : 20.0, speed : 180 },
5          { name : "L31", count : 10, blood : 15.0, speed : 180 }
6      ]
7  }
```

第3组怪物总共有15个，第1队有5个怪物，它们的怪物名字是L21，血量为20.0，移动速度为180。第2队有10个怪物，怪物名字为L31，血量为15.0，移动速度也为180。同样，经过`_loadMonsterData()`函数处理之后，第3组怪物的数据`_monsterDataArray[2]`的长度就是15，前5个数据保存的是第1队怪物的数据，后面10个保存的是第2队怪物的数据，如图17-8所示。

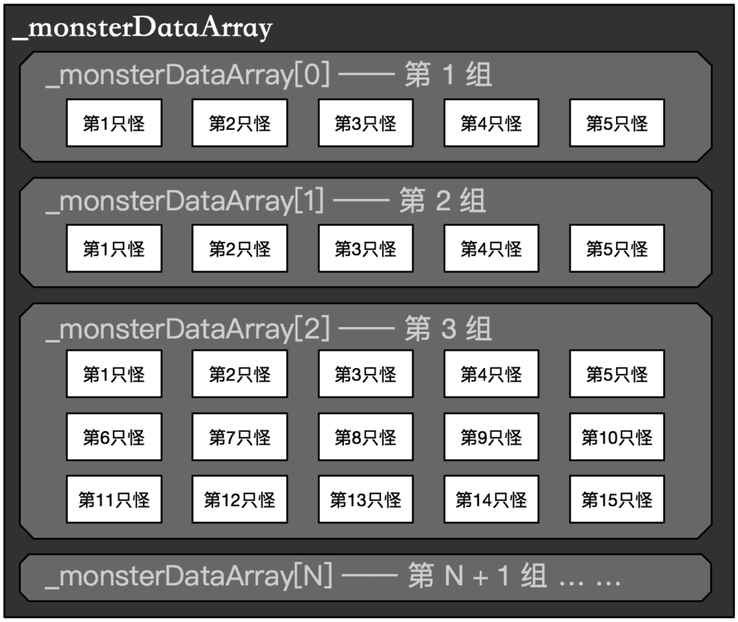


图17-8　`_monsterDataArray`的数据情况

或者，你也可以在调用`_loadMonsterData()`函数之后，直接调用`cc.log(this._monsterDataArray)`函数将`_monsterDataArray`对象打印出来，此时也可以在Chrome Console中看到`_monsterDataArray`的数据情况。

3. 获取下一个怪物数据

`_loadMonsterData()`函数中的第12行调用了`this._getNextMonsterData()`函数，该函数会返回下一个当前关卡的下一个怪物。例如，第1关总共有60个怪物，那么第1次调用`_getNextMonsterData()`函数，将返回第1个怪物数据，第50次调用`_getNextMonsterData()`

函数，将返回第50个怪物数据，代码如下：

```
1  //获取下一个怪物数据
2  _getNextMonsterData : function(){
3      if (this.isMonsterGetFinish == true) {
4          cc.warn("GameManager._getNextMonsterData() : 所有怪物数据已经获取完毕！");
5          return;
6      }
7
8      var teamData = this.monsterGroup[this._groupIndex].team[this._teamIndex];
9      var monsterData = {};
10     monsterData.group = this._groupIndex;
11     monsterData.name = teamData.name;
12     monsterData.blood = teamData.blood;
13     monsterData.speed = teamData.speed;
14     monsterData.index = this._teamMonsterIndex;
15
16     this._teamMonsterIndex++;
17     // 是否进入到下一队
18     if (this._teamMonsterIndex > this._teamMonsterCount) {
19         this._enterNextTeam();
20     }
21
22     return monsterData;
23 }
```

在上述代码中，第8行根据_groupIndex和_teamIndex下标，在monsterGroup数组中获取出对应的怪物数据，每获取一次，_teamMonsterIndex执行加一操作。当_teamMonsterIndex大于_teamMonsterCount时，就应该进入到下一队去了。_enterNextTeam()函数的代码如下：

```
1  // 进入到下一队
2  _enterNextTeam : function(){
3      this._teamMonsterIndex = 0;
4      this._teamIndex++;
5      // 进入下一组
6      if (this._teamIndex > this._teamCount) {
7          this._enterNextGroup();
8      }
9      // 进入到下一队
10     else{
11         this._teamMonsterCount = this.monsterGroup[this._groupIndex].team[this.
           _teamIndex].count - 1;
12     }
13 }
```

在进入到下一队时，需要将_teamMonsterIndex的下标重置为0，表示又从下一队的第1只怪物开始获取，并且当前队伍的下标要执行加一操作，表示进入到下一队了。

最后，第6行到第12行判断是否还有下一队，如果没有，则进入到下一组，如果有，则重新设置当前队伍总的怪物数量的值。

第7行执行进入下一组操作，_enterNextGroup()函数的代码如下：

```
1  // 进入到下一组
2  _enterNextGroup : function(){
3      this._groupIndex++;
4      // 添加完毕
5      if (this._groupIndex > this.maxGroup) {
6          this.isMonsterGetFinish = true;
7          return;
8      }
9      this._teamIndex = 0;
10     this._teamCount = this.monsterGroup[this._groupIndex].team.length - 1;
11     this._teamMonsterIndex = 0;
12     this._teamMonsterCount = this.monsterGroup[this._groupIndex].team[this.
       _teamIndex].count - 1;
13 }
```

毫无疑问，首先需要执行组下标加一操作（见第3行），表示进入下一组了。然后第5行到第8行判断是否所有的怪物都获取完毕，如果获取完毕，则标记isMonsterGetFinish的值为true，如果还没完全获取出所有的怪物数据，则继续重新设置_teamIndex和_teamCount等相关值，这保证了下一次调用_getNextMonsterData()时，能正常获取出对应的怪物数据。获取下一只怪物的数据的整个过程如图17-9所示。

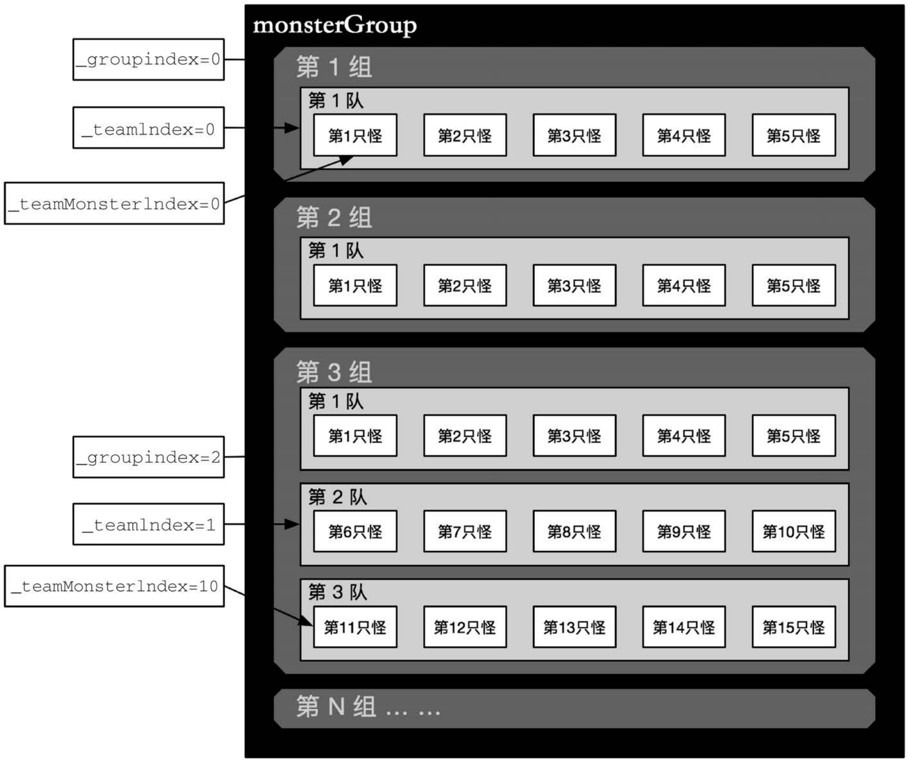


图17-9　获取下一只怪物数据的示意图

17.2.3　弹出下一组怪物数据

当所有的数据处理完毕之后，我给GameManager对象编写了一个popNextMonsterGroup-Data函数，用来弹出下一组怪物的数据。这样在游戏玩法场景中，每次出怪，只要调用GameManager.popNextMonsterGroupData()函数便可以得到对应组的怪物数据。该函数的代码如下：

```
1  //弹出下一组怪物数据
2  popNextMonsterGroupData : function() {
3      var groupData = [];
4      if (this.group <= this.maxGroup) {
5          this.group++;
6          groupData = this._monsterDataArray[0];
7          this._monsterDataArray.splice(0, 1);
8
9          // [抛出事件]组别更新
10         var event = new cc.EventCustom(jf.EventName.GP_UPDATE_GROUP);
11         event.setUserData({
12             group : this.group
13         });
14         cc.eventManager.dispatchEvent(event);
15     }else{
16         groupData = [];
17     }
18     return groupData;
19 }
```

需要特别注意的是第5行，我在这里更新了当前组的编号。在当前组还不是最后一组的情况下，每次调用popNextMonsterGroupData()函数都会使得group加一。所以，当获取第1组敌人数据的时候，group的值已经是1了。需要注意的是，在弹出下一组怪物数据之后，如果需要通过当前组作为相关数组的下标，则正确的下标值应该是group - 1。

popNextMonsterGroupData函数执行的是弹出数据操作所以，第6行每次得到的怪物组数据都是_monsterDataArray数组的第一个对象。然后在第7行中，将_monsterDataArray数组的第一个对象“剪掉”。这两行代码，保证了当前函数每次是从_monsterDataArray执行弹出数据操作。

第9行到第14行抛出了一个组别更新的事件，事件名称jf.EventName.GP_UPDATE_GROUP定义在define.js文件中。这个事件，将在游戏玩法场景中的UI层去实现监听。当组别更新时，UI层便可以收到组别更新的事件，从而更新当前组的文本显示。

说明　Carrot项目中所有的自定义事件名称都定义在define.js文件中，后面不再说明。

此时，你可以在main.js文件中的cc.game.onStart()函数里测试下GameManager对象，例如我加载了第一关的数据，并且弹出了3组怪物数据，代码如下：

```
1   GameManager.loadLevelData(0);
2   cc.log(GameManager.popNextMonsterGroupData());
3   cc.log(GameManager.popNextMonsterGroupData());
4   cc.log(GameManager.popNextMonsterGroupData());
```

再运行项目，可以在Chrome Console中看到输出日志如图17-10所示。

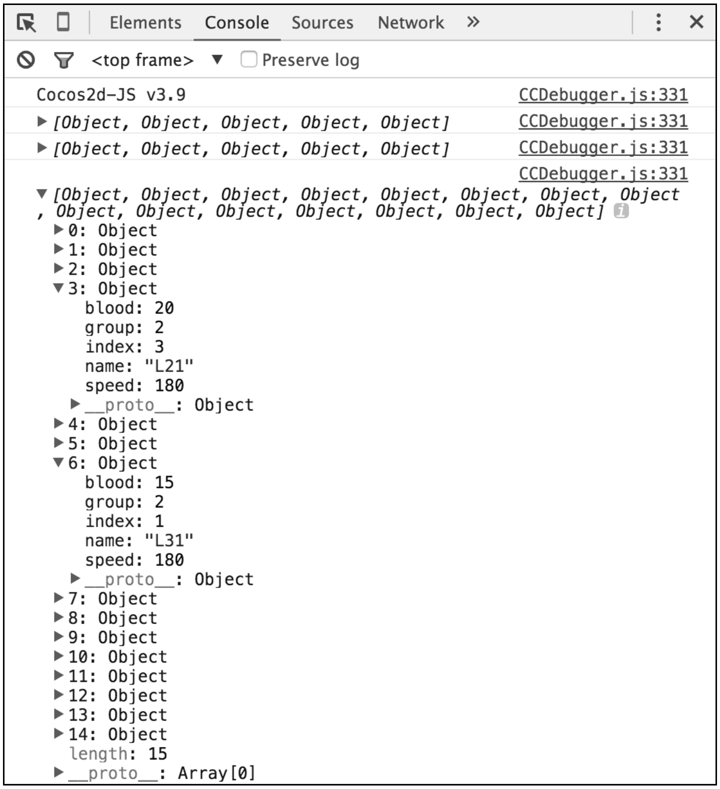


图17-10　GameManager弹出的3组怪物数据

17.3　游戏玩法场景开发

我将游戏玩法场景定义为GamePlayScene，它里面包含了4个显示的层，分别是背景层（GPBackgroundLayer）、游戏玩法层（GPMainLayer）、UI层（GPUILayer）和一个隐藏的菜单层（GPMenuLayer）。最底层的是背景层，中间层是玩法层，最上面的是UI层，而菜单层则是点击UI层中的菜单按钮后创建出来的层。整个GamePlayScene场景开发完成的项目情况如图17-11所示。

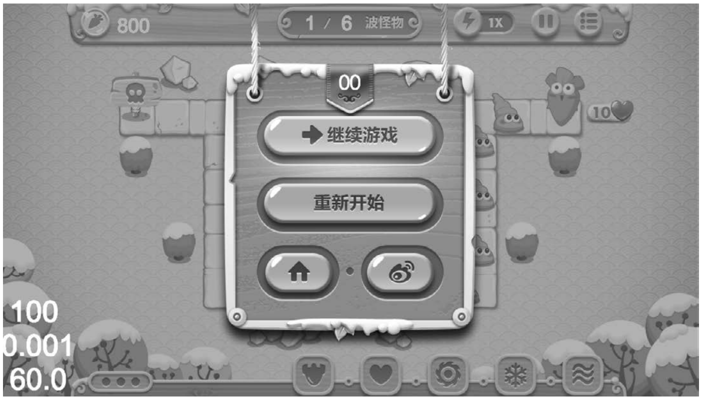


图17-11　GamePlay场景完成

17.3.1　GamePlayScene 开发

在src/Scene下新建GamePlay以及GamePlay/Layer等子文件夹，并在src/Scene/GamePlay文件夹中新建了GamePlay.js文件，然后在src/Scene/GamePlay/Layer路径下新建Background.js、Main.js、Menu.js以及UI.js文件来存放对应层的逻辑代码，完成之后的目录结构如图17-12所示。

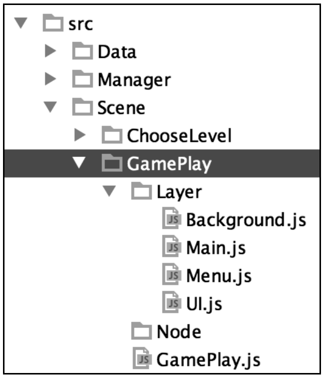


图17-12　GamePlayScene场景相关代码的组织结构

然后在GamePlay.js文件中编写游戏玩法场景`GamePlayScene`。与主页面场景`MainMenuScene`类似，我定义了`loadBackgroundLayer()`、`loadMainLayer()`、`loadUILayer()`以及`loadMenuLayer()`函数来加载对应的层，详见GamePlay.js文件。

在`GamePlayScene`中，我给它编写了一个`registerEvent()`函数来注册事件监听器，其代码如下：

```
1  // 注册[事件]
2  registerEvent : function(){
```

```
3       // [事件监听器]创建菜单层
4       var onCreateMenuLayerListener = cc.EventListener.create({
5           event        : cc.EventListener.CUSTOM,
6           target       : this,
7           eventName    : jf.EventName.GP_CREATE_MENU_LAYER ,
8           callback     : this.onCreateMenuLayer
9       });
10      cc.eventManager.addListener(onCreateMenuLayerListener, this);
11
12      // [事件监听器]移除菜单层
13      var onRemoveMenuLayerListener = cc.EventListener.create({
14          event        : cc.EventListener.CUSTOM,
15          target       : this,
16          eventName    : jf.EventName.GP_REMOVE_MENU_LAYER ,
17          callback     : this.onRemoveMenuLayer
18      });
19      cc.eventManager.addListener(onRemoveMenuLayerListener, this);
20  }
```

上面的代码给GamePlayScene注册了两个事件监听器，监听的事件分别是创建菜单层事件和移除菜单层事件，事件的回调函数为onCreateMenuLayer()和onRemoveMenuLayer()，这两个函数实现了菜单层的创建和删除。毫无疑问，这两个事件都是在UI层（GPUILayer）中被触发的。这两个函数的实现代码如下：

```
1  onCreateMenuLayer : function(event) {
2      var self = event.getCurrentTarget();
3      self.loadMenuLayer();
4  },
5  onRemoveMenuLayer : function(event){
6      var self = event.getCurrentTarget();
7      self.removeChild(self.menuLayer);
8  }
```

17.3.2 背景层 GPBackgroundLayer 开发

《保卫萝卜2》中有个主题概念，每个主题的背景图片都是不一样的。关卡主题也都配置在LevelData中，主题可以通过GameManager.getThemeID()函数获取出来。GPBackground-Layer的代码如下：

```
1  var GPBackgroundLayer = cc.Layer.extend({
2      onEnter : function () {
3          this._super();
4          // 加载[背景]
5          this.loadBackgound();
6      },
7      loadBackgound : function(){
8          var themeID = GameManager.getThemeID();
9          var node = new cc.Sprite("res/GamePlay/Theme/Theme" + themeID + "/BG0/BG"
           + themeID + ".png");
10         this.addChild(node);
11         node.setPosition(cc.winSize.width / 2, cc.winSize.height / 2);
```

```
12     }
13 });
```

17.3.3 UI 层 GPUILayer 开发

在UI层，主要的内容就是顶部的一些工具栏和底部的一些图标等，所以GPUILayer中大部分的代码都是处理一些简单的UI控件，并没有比较深入的内容。故此，我不打算一一讲解，大家可以阅读随书源码。

你应该记得，在GameManager对象的popNextMonsterGroupData()函数中会抛出一个组更新的事件，这个事件是在GPUILayer中实现监听的。当组别更新时，UI显示的当前组的文本内容也需要更新，所以，应该在GPUILayer中注册这一事件监听，相关代码如下：

```
1  var GPUILayer = cc.Layer.extend({
2      // ......
3      onEnter : function(){
4          this._super();
5          // ......
6          // 注册[事件]
7          this.registerEvent();
8      },
9      registerEvent : function(){
10         // 组更新
11         var onUpdateGroupListener = cc.EventListener.create({
12             event       : cc.EventListener.CUSTOM,
13             target      : this,
14             eventName   : jf.EventName.GP_UPDATE_GROUP,
15             callback    : this.onUpdateGroup
16         });
17         cc.eventManager.addListener(onUpdateGroupListener, this);
18     },
19     onUpdateGroup : function(event){
20         var group = event.getUserData().group;
21         var self = event.getCurrentTarget();
22         self.groupText.setString(group + "");
23     }
24 });
```

在第15行中可以看到，组更新事件监听器的事件回调函数为this.onUpdateGroup，该函数接收一个事件对象参数event，该参数携带的用户数据中保存了当前是哪一组，这在第20行中被获取出来，然后第22行更新了组的文本显示。需要说明的是，当前组已经在GameManager.popNextMonsterGroupData()函数中执行过加一操作了，所以这里无需再加一了。

此时可以在main.js文件中的cc.game.onStart()函数里编写如下代码：

```
1  cc.LoaderScene.preload(g_gamePlay_resources, function () {
2      GameManager.loadLevelData(0);  // 加载第一关的数据
3      cc.director.runScene(new GamePlayScene());
4  }, this);
```

需要说明的是，g_gamePlay_resources为游戏场景所需要的素材数组。这样做的方式使得游戏在启动的时候，只从服务器上下载游戏玩法场景中所需的资源，绕过了游戏主页面、游戏关卡选择页面，直接进入到GamePlayScene，节省了一些等待时间。

此时，再运行一下项目，可以看到效果如图17-13所示。

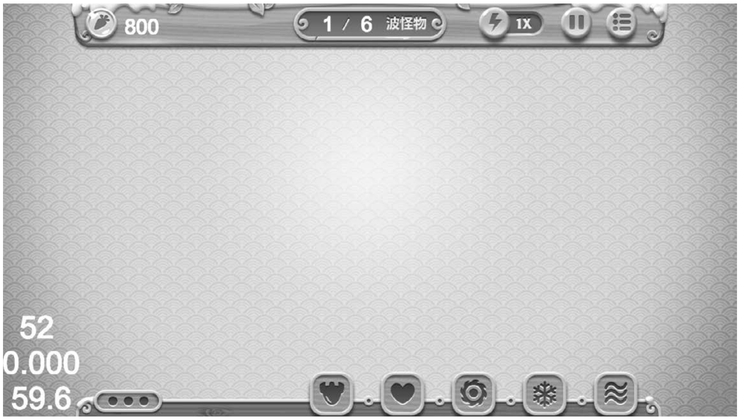


图17-13　游戏场景背景层和UI层开发完成

17.3.4　菜单层 GPMenuLayer 开发

前面提到过，GamePlayScene中有一个隐藏的菜单层，这个层是通过点击GPUILayer中的菜单按钮后所创建出来的。这个按钮的创建代码如下：

```
1  // 加载[菜单按钮]
2  loadMenuButton : function () {
3      var node = new ccui.Button();
4      // ......
5      node.addTouchEventListener(function(sender, type) {
6          switch (type) {
7              case ccui.Widget.TOUCH_ENDED:     // 触摸事件结束时响应
8                  var event = new cc.EventCustom(jf.EventName.GP_CREATE_MENU_
                   LAYER);
9                  cc.eventManager.dispatchEvent(event);
10                 break;
11         }
12     }.bind(this));
13 }
```

第8行创建了一个名字为jf.EventName.GP_CREATE_MENU_LAYER的创建菜单层自定义事件，然后第9行通过事件管理对象cc.eventManager分发出去。因为我们在GamePlayScene.registerEvent()函数中注册了这一事件的监听，所以当GPUILayer中的菜单被点击后，GamePlayScene收到事件，从而调用onLoadMenuLayer()函数创建出GPMenuLayer层，并添加到GamePlayScene中。

GPMenuLayer中的代码，无非也就是一些UI控件的组合（详见Menu.js），但是这里我提一下打开GPMenuLayer之后，游戏暂停和恢复的设计，相关代码如下：

```
1  var GPMenuLayer = ccui.Layout.extend({
2      onEnter : function(){
3          this._super();
4          cc.director.pause();    // 导演暂停
5          // ......
6      },
7      onExit : function(){
8          cc.director.resume();   // 导演恢复
9          this._super();
10     }
11 });
```

第2行到第6行通过在GPMenuLayer的onEnter()函数中调用cc.director.pause()函数实现了游戏暂停，然后在onExit()函数中又调用了cc.director.resume()函数恢复了游戏。游戏的暂停和恢复放在GPMenuLayer生命周期的进入函数onEnter()以及退出函数onExit()中是一个不错的选择。

此时，再运行项目，点击UI层中的菜单按钮，可以看到页面如图17-14所示。至此，除了游戏玩法层（GPMainLayer）之外，其余的层都开发完毕了。



图17-14　菜单层

17.3.5　游戏玩法层（GPMainLayer）

GPMainLayer的开发，是Carrot的重点，也是难点。在开发之前，我先把GPMainLayer中要做的主要事情罗列出来，具体如下。

- 根据配置加载怪物移动路径。
- 根据配置加载怪物。

- 根据配置加载障碍物。
- 根据配置加载起点和终点。
- 判断当前位置是否可以创建炮塔，若可以则创建，若不行，则给出警告。
- 子弹和怪物的碰撞检测。
- 游戏胜利或失败的场景跳转。

有了任务清单之后，开发的逻辑就变得清晰一些了，现在一件一件来。首先，先给GPMainLayer定义出相关的属性，代码如下：

```
1  var GPMainLayer = cc.Layer.extend({
2      tiledMap            : null, // 瓦片地图
3      tileSize            : null, // 瓦片大小
4      roadPointArray      : [],   // 怪物路径
5      ZOrderEnum          : {},   // 对象层级枚举
6
7      carrot              : {},   // 萝卜[对象]
8      carrotHpBg          : {},   // 萝卜[血量背景]
9      carrotHpText        : {},   // 萝卜[血量]
10
11     tiledMapRectArray   : [],   // 瓦片地图区域[二维区域]
12     tiledMapRectArrayMap: [],   // 瓦片地图区域映射
13     tiledMapRectMapEnemu: {},   // 瓦片地图区域映射枚举
14     touchWarningNode    : null, // 触摸警告节点
15     towerPanel          : null, // 构建塔的面板
16
17     currGroupCreatedMonsterCount : 0,   // 当前组已经创建的怪物数量
18     currGroupCreatedMonsterSum   : 0,   // 当前组怪物总数量
19     onEnter: function () {
20         this._super();
21         // ......
22     }
23 });
```

这些属性，你先大概浏览一下，保证脑中有个概念即可，后面会接触到。

接下来，我们先把怪物的移动路径创建出来。

1. 加载怪物移动路径

在《保卫萝卜2》中，怪物的移动路径设定在对应的路径图片中，所以程序只需要获取出当前的主题和关卡，就能得到对应的路径图片。加载怪物的移动路径图片的代码比较简单，具体如下：

```
1  // 加载[路径背景]
2  loadPath : function(){
3      var themeID = GameManager.getThemeID();
4      var level = GameManager.getLevel() + 1;
5      var node = new cc.Sprite("res/GamePlay/Theme/Theme" + themeID + "/BG" + level
       + "/Path" + level + ".png");
6      this.addChild(node);
7      node.setPosition(cc.winSize.width / 2, cc.winSize.height / 2);
8  }
```

可以看到，第4行获取出了当前的关卡，但是执行了加一操作，这是因为图片资源命名从1开始，但是关卡数从0开始。

第5行根据主题和关卡数找到了对应的路径图片，并创建出了精灵节点。第7行把这个路径图片定位在当前场景的正中间。此时，再运行一下项目，可以看到效果如图17-15所示。

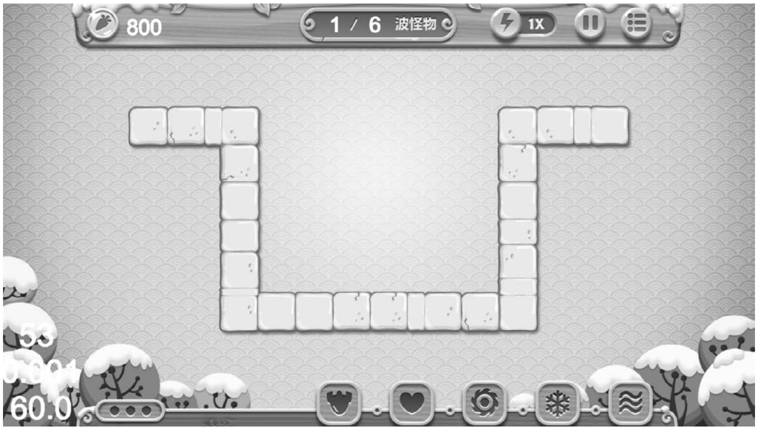


图17-15　怪物移动路径

2. 关卡配置数据——瓦片地图加载

在《保卫萝卜2》项目中，美术给出了每一个关卡障碍物等的示意图（见res/GamePlay/Theme/Theme1/BG[x]文件夹，其中x表示关卡数），例如第一关的场景情况如图17-16所示。

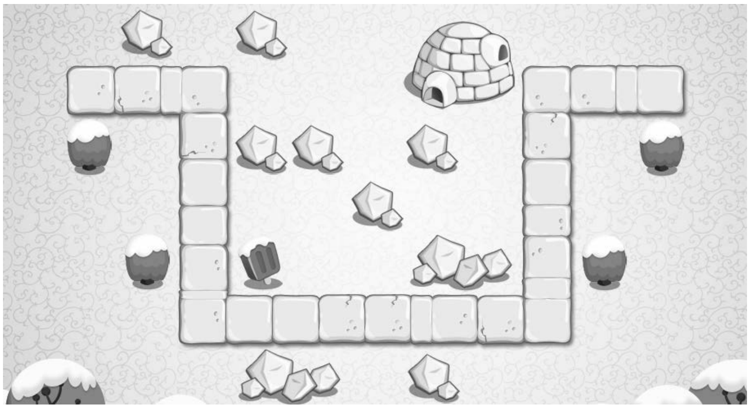


图17-16　第一关道路和障碍物情况

我们可以直接把这张图片拖进Tiled Map Editor编辑器中，然后像之前标记选择关卡按钮的位置那样，标记出怪物的移动路径坐标点以及其他障碍物的位置。那么，现在就来试试看如何在Tiled Map Editor编辑器中编辑这些关卡配置吧。

在《保卫萝卜2》中，小障碍物的大小为70×70。也就是说，我们新建瓦片地图的时候，每一个瓦片块的大小也应当设置为70×70。另外，图17-16的大小为910×490，宽和高都除以70之后，可以得到要新建的瓦片地图的地图大小应该为13×7。

接着要进行的操作步骤如下所示。

(1) 将瓦片地图文件保存在res/GamePlay/Theme/Theme1/BG1文件夹下，并将其命名为Level1。

(2) 将res/GamePlay/Theme/Theme1/BG1/1.png图片拖进瓦片地图编辑器的图块面板中。

(3) 修改默认图层的名称为bg。

(4) 选中bg图层，选中“图章刷”工具，选择图块1中所有的瓦片，在bg层中“印”出来。

(5) 创建一个对象层，并将其命名为road，用来保存怪物的移动路径。

接下来，标记出怪物的移动路径。选中road对象层，再选中“插入矩形”工具，然后在地图编辑区域中插入矩形，从而标记出坐标点，如图17-17所示。

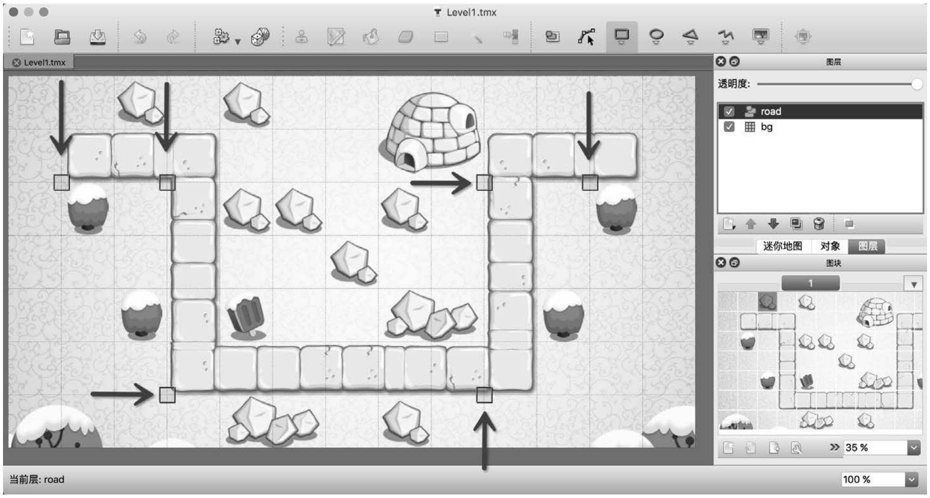


图17-17　怪物移动路径标记

这里需要特别注意的是，插入矩形的时候一定要有顺序，按照怪物的移动路径一个一个标记出来，否则在代码中获取出来的路径将会出现异常。

有了上面的坐标标记之后，怪物每次在移动的时候，只要获取当前坐标标记点和下一个拐角的坐标标记点，然后执行一个moveTo动作便可以实现根据路径移动了。

然后，我们在代码中将瓦片地图读取出来，相关代码如下：

```
1  // 加载[瓦片地图，主要是一些关卡信息配置]
2  loadTiledMap : function(){
3      var themeID = GameManager.getThemeID();
```

```
4       var level = GameManager.getLevel() + 1;
5       var node = new cc.TMXTiledMap("res/GamePlay/Theme/Theme" + themeID + "/BG" +
        level + "/Level" + level + ".tmx");
6       this.addChild(node);
7       this.tiledMap = node;
8       this.tileSize = node.getTileSize();
9       node.x = (cc.winSize.width - node.width) / 2;
10      node.y = (cc.winSize.height - node.height) / 2;
11      node.setVisible(false);
12
13      // 设置[所有对象组]坐标偏移量
14      var groups = this.tiledMap.getObjectGroups();
15      var group = null;
16      var offsetX = (cc.winSize.width - this.tiledMap.width) / 2;
17      var offsetY = (cc.winSize.height - this.tiledMap.height ) / 2;
18      var finalOffsetX = 0;
19      var finalOffsetY = 0;
20      var groupName = "";
21      for (var i = 0; i < groups.length; i++) {
22          group = groups[i];
23          groupName = group.getGroupName();
24
25          if (groupName == "road") {
26              finalOffsetX = offsetX + this.tileSize.width / 2;
27              finalOffsetY = offsetY + this.tileSize.height / 2;
28          }else{
29              cc.warn("GPMainLayer.loadTiledMap(): " + groupName + "对象组的坐标未调
                整");
30          }
31
32          group.setPositionOffset(cc.p(finalOffsetX, finalOffsetY));
33      }
34 }
```

在上述代码中，第3行到第5行获取出当前主题和关卡，然后根据主题和关卡创建出cc.TMXTiledMap节点。

第8行保存了瓦片地图的地图大小，其值为cc.size(13, 7)。

第9行到第10行设置瓦片地图的坐标为当前场景的正中心。

第11行中，我把瓦片地图隐藏了，因为我只是需要瓦片地图对象层中的配置数据。

另外，在瓦片地图编辑器中，对象层中参考的原点坐标为当前地图的左下角，但是瓦片地图被添加到当前场景的正中心之后，就会产生一定的位置偏移。第13行到第32行就是计算每个对象层位置的偏移量，最终把位置的偏移量保存在对应的group中（见第32行）。

3. 怪物生成

调整完怪物移动的路径标记坐标后，便可以创建怪物，实现怪物移动了。

首先，需要把瓦片地图中road对象层里面所有的标记获取出来，然后加上road道路对象层

的坐标偏移量，最后将坐标点保存在roadPointArray数组中，代码如下：

```
1  // 加载[路径坐标]
2  loadRoadPointArray : function(){
3      this.roadPointArray = [];
4      var roadGroup = this.tiledMap.getObjectGroup("road");
5      var roads = roadGroup.getObjects();
6      for (var i = 0; i < roads.length; i++) {
7          this.roadPointArray.push(cc.p(roads[i].x + roadGroup.getPositionOffset().
           x , roads[i].y + roadGroup.getPositionOffset().y));
8      }
9  }
```

接着，再编写一个loadNextGroupMonster()函数。每次调用这个函数时，都会创建出当前组的怪物，其代码如下：

```
1  // 加载[下一组怪物]
2  loadNextGroupMonster : function(){
3      if (GameManager.getGroup() > GameManager.getMaxGroup()) {
4          cc.log("GPMainLayer.loadNextGroupMonster() : 怪物添加完毕");
5          return;
6      }
7      GameManager.currMonsterDataPool = GameManager.popNextMonsterGroupData();
8      GameManager.currMonsterPool[GameManager.getGroup() - 1] = [];
9
10     this.currGroupCreatedMonsterCount = 0;
11     // 怪物总数统计
12     this.currGroupCreatedMonsterSum = GameManager.getCurrGroupMonsterSum();
13
14     var groupDelay = cc.delayTime(GameManager.getGroupInterval());
15     // 延迟时间
16     var enemyDelay = cc.delayTime(GameManager.getEnemyInterval());
17     var callback = cc.callFunc(this.createMonster.bind(this));
18     var createMonsterAction = cc.sequence(enemyDelay.clone(), callback).repeat
       (this.currGroupCreatedMonsterSum);
19     var finalAction = cc.sequence(groupDelay, createMonsterAction);
20     this.runAction(finalAction);
21 }
```

在上述代码中，第7行弹出了一组怪物数据，并保存在GameManager.currMonster-DataPool数组中。

第8行重置了当前组的怪物节点池。

第9行重置了当前组已经创建的怪物数量。

第12行获取当前组总共有多少只怪物（具体代码可见GameManager.getCurrGroup-MonsterSum()函数），并将结果赋值给currGroupCreatedMonsterSum参数，这个结果作用于第18行，用来重复执行多少次创建怪物的复合函数。

第17行创建了一个函数动作。createMonster()函数每次会从GameManager.curr-MonsterDataPool数组中获取下一个怪物数据，然后根据这个数据创建出怪物，其代码如下：

```
1  // 创建[怪物]
2  createMonster : function(){
3      var data = GameManager.currMonsterDataPool[0];
4      // 创建怪物数量+1
5      this.currGroupCreatedMonsterCount++;
6
7      var monsterData = {
8          road : this.roadPointArray,
9          speed : data.speed,
10         index : data.index
11     };
12
13     var namePrefix = data.name.substring(0, data.name.length - 1);
14     var fileNamePrefix = "Theme" + GameManager.getThemeID() + "/Monster/" +
       namePrefix;
15     var fileName = "#" + fileNamePrefix + "1.png";
16     var node = new Monster(fileName, monsterData, fileNamePrefix);
17     this.addChild(node, this.ZOrderEnum.MONSTER);
18     GameManager.currMonsterPool[GameManager.getGroup() - 1].push(node);
19     node.setPosition(this.roadPointArray[0]);
20     node.run();
21
22     // 删除第一个数据
23     GameManager.currMonsterDataPool.splice(0, 1);
24 }
```

在上述代码中，第13行剪切掉了名字的最后一个字符。例如，L21经过剪切之后，得到L2前缀；又比如SF21经过剪切之后，得到SF2前缀。获取这个前缀的目的是将此前缀传递给怪物类（`Monster`），然后在怪物类内部，便可以根据此前缀拼接出对应的帧动画名字，因为每一个怪物都有3帧，例如胖怪的帧图片如图17-18所示。



图17-18　胖怪的序列帧图片

第16行根据对应的主题和关卡，创建出对应的怪物，并把怪物数据`monsterData`传到`Monster`构造函数中。

第18行将创建出来的怪物保存在`GameManager.currMonsterPool`数组中。

第23行删除了`GameManager.currMonsterDataPool`数组的第一个对象。

注意第20行，这里执行了`node.run()`函数，调用此函数后，怪物便可以一边播放着动画，一边跑起来。

显然，`run`函数被定义在`Monster`类（位于src/Scene/GamePlay/Node/Monster/Monster.js文件里）中，接下来就来看看`Monster`类是如何实现的。

首先，我给Monster类定义了几个属性，代码如下：

```
1  var Monster = cc.Sprite.extend({
2      road     : [],    // 移动路径
3      data     : {},    // 数据
4      speed    : 0,     // 速度
5      index    : 0,     // 索引
6      roadIndex        : 0,     // 当前移动路径的前缀
7      fileNamePrefix   : "",    // 帧前缀
8      ctor : function(fileName, data, fileNamePrefix){
9          this._super(fileName);
10         // ......
11     }
12 });
```

然后，我又给Monster编写了一个loadProperty(data, fileNamePrefix)函数，用来配置初始属性，代码如下：

```
1  // 加载[属性配置]
2  loadProperty : function(data, fileNamePrefix){
3      cc.assert(data.speed, "Monster.loadProperty(): 速度不能为空！");
4      cc.assert(data.road, "Monster.loadProperty(): 移动路径不能为空！");
5      cc.assert(data.index >= 0, "Monster.loadProperty(): 索引不能为空！");
6      cc.assert(fileNamePrefix, "Monster.loadProperty(): 文件名前缀不能为空！");
7
8      this.data = data;
9      this.speed = data.speed;
10     this.road = data.road;
11     this.index = data.index;
12     this.fileNamePrefix = fileNamePrefix;
13 }
```

在loadProperty函数中，添加了几个断言，用来保证属性被正确赋值。有了速度、移动路径等属性后，怪物便可以运行moveTo动作来实现跑到终点。接着，给Monster添加runNextRoad函数，用来让怪物跑到下一个标记点上，直到跑到终点，其代码如下：

```
1  // 跑到下一个标记点上
2  runNextRoad : function(){
3      // 转方向
4      if (this.road[this.roadIndex].x <= this.road[this.roadIndex + 1].x) {
5          this.setFlippedX(false);
6      }else{
7          this.setFlippedX(true);
8      }
9      var distance = cc.pDistance(this.road[this.roadIndex], this.road[this.
       roadIndex + 1]);
10     var time = distance / this.speed;
11     var moveTo = cc.moveTo(time, this.road[this.roadIndex + 1]);
12     var callback = cc.callFunc(function(){
13         if (this.roadIndex < this.road.length - 1) {
14             this.runNextRoad();
15         }else{
16             // 吃到萝卜事件抛出
```

```
17                var event = new cc.EventCustom(jf.EventName.GP_MONSTER_EAT_CARROT);
18                event.setUserData({
19                    target : this
20                });
21                cc.eventManager.dispatchEvent(event);
22            }
23        }.bind(this));
24        var seq = cc.sequence(moveTo, callback);
25        this.runAction(seq);
26        this.roadIndex++;
27    }
```

第3行到第8行里面有个小技巧，就是当你在瓦片地图编辑器中标记移动路径的时候，若希望怪物的脸转个方向，便可把X轴坐标向左或向右偏移1个像素。在程序中，对比下当前怪物所在的路径点和下一个路径点的X轴坐标，若下一个标记点的X轴坐标小于当前标记点的X轴坐标，那么调用`this.setFlippedX(true)`函数，即可实现了怪物脸的朝向。

另外，值得注意的是第13行，它判断是否移动到最后一个标记点，若不是，则执行第14行，继续调用`runNextRoad()`函数让怪物跑到下一个标记点上，直到怪物到达终点之后，抛出怪物吃到萝卜的事件，并把当前怪物对象作为事件携带数据，一并抛出，然后接收事件的层便可以通过事件携带数据将怪物移除。毫无疑问，这一事件将在`GPMainLayer`中被注册监听。

此时，再运行下项目，可以看到结果如图17-19所示。

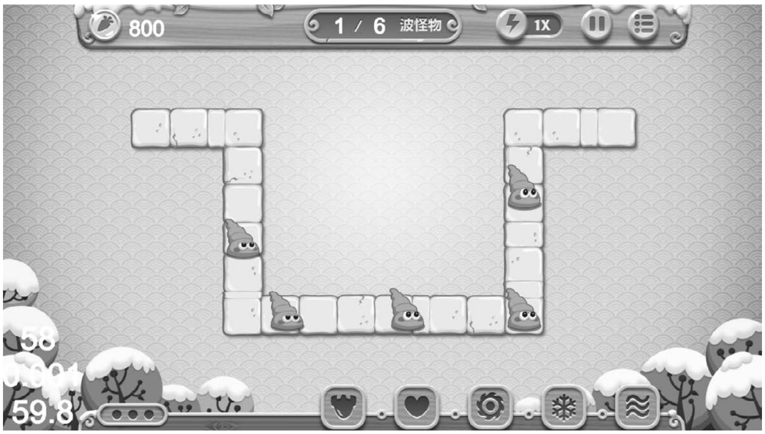


图17-19　创建出第一组怪物

4. 开始路标以及萝卜的开发

在怪物到达终点的时候，会抛出一个吃到萝卜的事件，但是此时我们还没有将萝卜和开始路标添加进来，接下来就来完成开始路标以及萝卜的开发。

实际上，开始路标和萝卜的坐标位置就是瓦片地图`road`对象层中第一个对象和最后一个对象的坐标，但是我并不打算利用这两个值，因为这会加大移动路径和萝卜的耦合度。这里我在Tiled

Map Editor编辑器中新建了一个start_end对象层，然后在道路的起点和终点标记出了对应的位置，这一步的操作和之前标记怪物的移动路径是一样的。

给瓦片地图添加start_end对象层后，也需要在GPMainLayer.loadTiledMap()函数中设置start_end对象层的坐标偏移量，相关代码见下面的第7行：

```
1  // 加载[瓦片地图，主要是一些关卡信息配置]
2  loadTiledMap : function(){
3      // ......
4      for (var i = 0; i < groups.length; i++) {
5          // ......
6          if (groupName == "road"
7              || groupName == "start_end") {
8              finalOffsetX = offsetX + this.tileSize.width / 2;
9              finalOffsetY = offsetY + this.tileSize.height / 2;
10         }else{
11             cc.warn("GPMainLayer.loadTiledMap(): " + groupName + "对象组的坐标未调
               整");
12         }
13         group.setPositionOffset(cc.p(finalOffsetX, finalOffsetY));
14     }
15 }
```

然后再给GPMainLayer编写loadStartAndEnd()函数，用来加载开始路标和终点萝卜。在loadStartAndEnd()函数中，又调用了loadStartFlag()函数，用来加载起始路标；调用了loadEndFlag()函数加载终点的萝卜。最后，调用loadCarrotHp()函数来加载萝卜的血量等相关节点，这些函数的实现详见随书源码。

现在，便可以在GPMainLayer中注册怪物吃到萝卜的事件监听了，代码如下：

```
1  // 注册[事件监听]
2  registerEvent : function(){
3      // [事件监听]怪物吃到萝卜
4      var onMonsterEatCarrotListener = cc.EventListener.create({
5          event       : cc.EventListener.CUSTOM,
6          target      : this,
7          eventName   : jf.EventName.GP_MONSTER_EAT_CARROT,
8          callback    : this.onMonsterEatCarrot
9      });
10     cc.eventManager.addListener(onMonsterEatCarrotListener, this);
11
12     // [事件监听]萝卜血量更新
13     var onUpdateCarrotBloodListener = cc.EventListener.create({
14         event       : cc.EventListener.CUSTOM,
15         target      : this,
16         eventName   : jf.EventName.GP_UPDATE_CARROT_BLOOD,
17         callback    : this.onUpdateCarrotBlood
18     });
19     cc.eventManager.addListener(onUpdateCarrotBloodListener, this);
20
21 }
```

可以看到，上面的代码注册了两个事件监听，分别是怪物吃到萝卜和萝卜血量更新，它们的事件回调函数分别是onMonsterEatCarrot(event)和onUpdateCarrotBlood(event)。这两个事件回调函数的逻辑代码如下：

```
1  // [事件]怪物吃到萝卜
2  onMonsterEatCarrot : function(event){
3      var self = event.getCurrentTarget();
4      // 萝卜扣血
5      GameManager.subtractCarrotBlood();
6      // 删除敌人
7      var monster = event.getUserData().target;
8      self.removeMonster(monster);
9      // 进入下一组
10     if (self.isNeedLoadNextGroup()) {
11         self.loadNextGroupMonster();
12     }
13 },
14 // [事件]更新萝卜血量
15 onUpdateCarrotBlood : function(event){
16     var self = event.getCurrentTarget();
17     var blood = event.getUserData().blood;
18     self.carrotHpText.setString(blood + "");
19 }
```

在上述代码中，onUpdateCarrotBlood(event)函数的内部实现比较简单，主要就是更新一下萝卜血量的文本显示。

而onMonsterEatCarrot(event)函数就显得复杂一些。首先，需要对萝卜进行扣掉一滴血的操作，这由GameManager.subtractCarrotBlood()函数来实现，该函数内部的实现代码如下：

```
1  // 萝卜每次扣一滴血
2  subtractCarrotBlood : function(){
3      this.carrotBlood = this.carrotBlood <= 0 ? 0 : this.carrotBlood - 1;
4      // 抛出血量更新事件
5      var event = new cc.EventCustom(jf.EventName.GP_UPDATE_CARROT_BLOOD);
6      event.setUserData({
7          blood : this.carrotBlood
8      });
9      cc.eventManager.dispatchEvent(event);
10
11     // 抛出游戏结束事件
12     if (this.carrotBlood == 0) {
13         var gameOverEvent = new cc.EventCustom(jf.EventName.GP_GAME_OVER);
14         gameOverEvent.setUserData({
15             isWin : false
16         });
17         cc.eventManager.dispatchEvent(gameOverEvent);
18     }
19 }
```

第3行通过三目运算保证了萝卜的血量永远不会小于0，然后每一次调用GameManager.

subtractCarrotBlood()函数，都会抛出一个血量更新事件，这一事件监听注册在我刚刚提到的GPMainLayer.registerEvent()函数中。此外，如果萝卜的血量等于0（见第12行），则抛出游戏结束事件，事件携带对象中标记了当前游戏为输了的状态（见第15行）。最后，第17行分发了游戏结束的事件。

现在，回到onMonsterEatCarrot()函数中，其中第7行获取出了当前怪物对象。第8行通过removeMonster(monster)函数将此怪物移除掉，该函数会去遍历GameManager.currMonsterPool数组，找到对象后，再通过removeMonsterByIndex(i, j)函数将怪物完全移除掉。removeMonster()和removeMonsterByIndex(i, j)函数的代码如下：

```
1  // 移除[怪物]
2  removeMonster : function(obj){
3      var monster = null;
4      for (var i = 0; i < GameManager.currMonsterPool.length; i++) {
5          for (var j = 0; j < GameManager.currMonsterPool[i].length; j++) {
6              monster = GameManager.currMonsterPool[i][j];
7              if (monster == obj) {
8                  this.removeMonsterByIndex(i, j);
9              }
10         }
11     }
12 },
13 // 根据数组下标删除怪物
14 removeMonsterByIndex : function(i, j){
15     this.removeChild(GameManager.currMonsterPool[i][j]);
16     GameManager.currMonsterPool[i].splice(j, 1);
17 }
```

在onMonsterEatCarrot()函数中做的最后一件事情就是通过isNeedLoadNextGroup()函数判断是否需要加载下一波怪物。isNeedLoadNextGroup()函数用于判断

GameManager.currMonsterPool数组中是否还有怪物存在，并且当前组已经创建的怪物数量（currGroupCreatedMonsterCount）是否等于当前组怪物的总数量（currGroup-CreatedMonsterSum），如果条件成立，则加载下一组怪物，代码如下：

```
1  // 判断是否需要进入到下一组
2  isNeedLoadNextGroup : function(){
3      var isNeed = false;
4      if (GameManager.currMonsterPool[GameManager.group - 1].length == 0
5          && this.currGroupCreatedMonsterCount == this.currGroupCreated-
           MonsterSum){
6          isNeed = true;
7      }
8      return isNeed;
9  }
```

此时，再运行项目，等怪物吃到萝卜，可以看到的效果如图17-20所示。

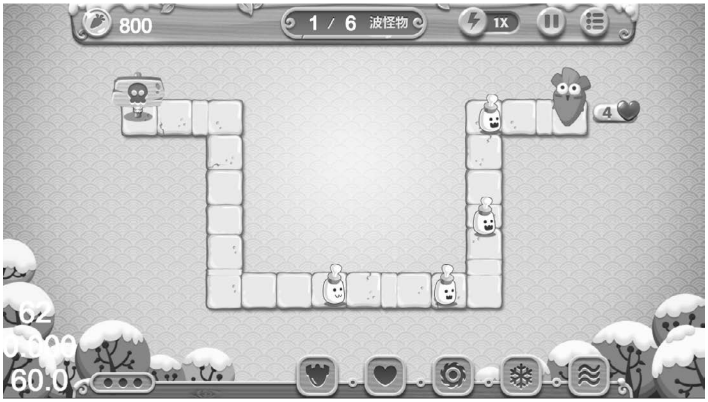


图17-20　怪物吃到萝卜

5. 障碍物加载

前面提到过，关卡中的障碍物也是使用Tiled Map Editor编辑器配置的，现在再给第一关的瓦片地图添加4个对象层，并将其分别命名为`small`（最小占1格的障碍物）、`little`（小障碍物，占左右两格的障碍物）、`big`（占4格的大障碍物）和`invalid`（不能点击区域）。然后在对应的图层中插入矩形，用来标记位置。其中，`small`和`invalid`对象层中标记的位置为格子的左下角，`little`对象层中障碍物标记的位置为第二个障碍物所在格子的左下角，`big`对象层中标记的位置为障碍物的正中心所在格子的左下角，如图17-21所示。

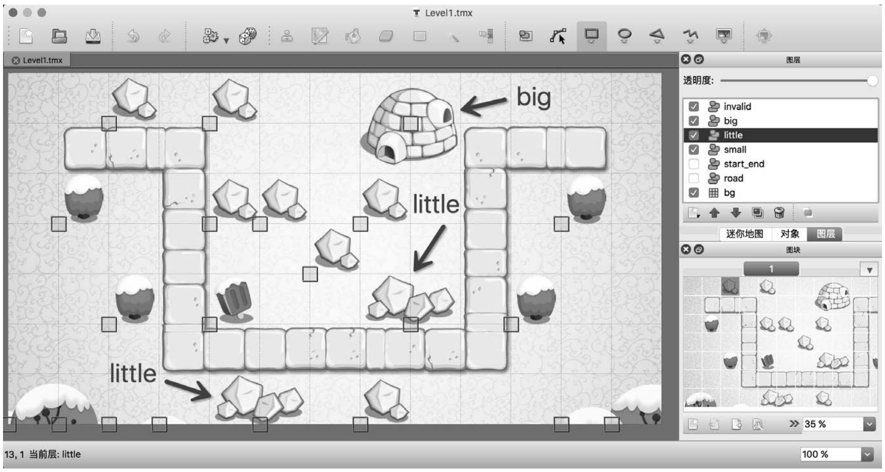


图17-21　障碍物标记点

但是，此时仅仅只是完成了各种障碍物位置的标记，对于当前位置是什么类型的障碍物，我们还没有标记，那么如何标记当前位置是什么类型的障碍物呢？实际上，这可以直接设定对象层中对象的名字，设定的名称也是对应障碍物的图片名称。图17-22演示的是`small`对象层中的标

记，其他对象层中障碍物的类型标记与其一致。

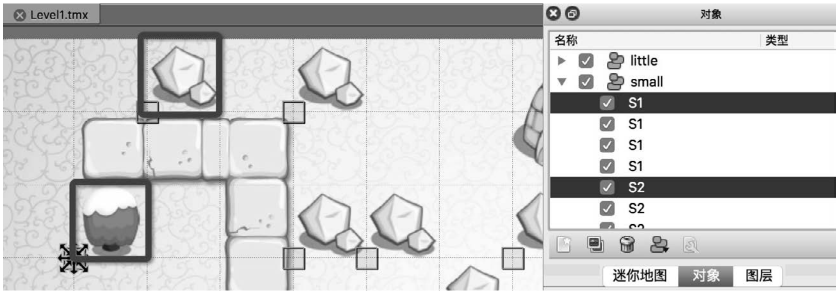


图17-22 障碍物类型标记

完成之后，回到代码中。在`GPMainLayer.loadTiledMap()`函数中设置每个对象层的坐标偏移量，代码如下：

```
1  // 加载瓦片地图，主要是一些关卡信息配置
2  loadTiledMap : function(){
3      // ......
4      for (var i = 0; i < groups.length; i++) {
5          group = groups[i];
6          groupName = group.getGroupName();
7
8          // 大障碍物[占4格]
9          if (groupName == "big") {
10             finalOffsetX = offsetX;
11             finalOffsetY = offsetY;
12         }
13         // 中等障碍物[占用左右2格]
14         else if (groupName == "little"){
15             finalOffsetX = offsetX;
16             finalOffsetY = offsetY + this.tileSize.height / 2;
17         }else if (groupName == "small"
18             || groupName == "road"
19             || groupName == "start_end"
20             || groupName == "invalid") {
21             finalOffsetX = offsetX + this.tileSize.width / 2;
22             finalOffsetY = offsetY + this.tileSize.height / 2;
23         }else{
24             cc.warn("GPMainLayer.loadTiledMap(): " + groupName + "对象组的坐标未调
               整");
25         }
26         group.setPositionOffset(cc.p(finalOffsetX, finalOffsetY));
27     }
28 }
```

可以看到，不同对象层设置的坐标偏移量不同，但是实际上它们都在做同一件事情——保证对象层中的对象坐标加上对象层的偏移量后所得到的坐标是当前障碍物的正确坐标。

当所有对象层坐标偏移量设置完成之后，便可以开始加载障碍物了。但是，要在程序中把这

些障碍物创建在对应的位置，并标记出当前位置是什么障碍物，可不像在Tiled Map Editor编辑器中那么简单。那么，应该怎么做呢？

首先，生成一个二维数组tiledMapRectArray。实际上，它是瓦片地图中瓦片格子区域的"映射"，保存地图中每个格子的区域（cc.rect）信息，如图17-23所示。

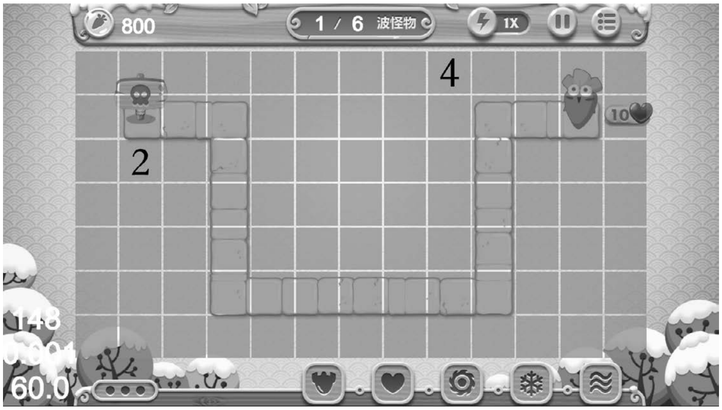


图17-23　障碍物类型标记

然后，还需要定义一个二维数组tiledMapRectArrayMap，这个数组是数组tiledMap-RectArray的状态标记。例如区域数组tiledMapRectArray中第5行第2列有一个小障碍物，那么只要设置tiledMapRectArrayMap[4][1]为一个约定的值，例如2，便成功地标记了区域中障碍物所在的位置以及类型了。其他的障碍物类似，例如你可以把占2格的障碍物标记为3，占用4格的大障碍物标记为4。

理清了逻辑之后，来看一下代码实现。显然 ，我们需要先配置区域数组tiledMapRect-Array，它在loadTileRect()函数中被赋值，代码如下：

```
1  // 加载[瓦片区域]
2  loadTileRect : function(){
3      var mapSize = this.tiledMap.getMapSize();
4      var nextPosX = (cc.winSize.width - this.tiledMap.width) / 2 + this.tileSize.
       width / 2;
5      var nextPosY = (cc.winSize.height - this.tiledMap.height) / 2 + this.tileSize.
       height / 2;
6      for (var i = 0; i < mapSize.height; i++) {
7          this.tiledMapRectArray[i] = [];
8          for (var j = 0; j < mapSize.width; j++) {
9              this.tiledMapRectArray[i][j] = cc.rect(
10                 nextPosX - this.tileSize.width / 2,
11                 nextPosY - this.tileSize.height / 2,
12                 this.tileSize.width,
13                 this.tileSize.height);
14             // 测试节点
15             var node = new cc.Sprite();
```

```
16              this.addChild(node, 100);
17              node.setTextureRect(cc.rect(0, 0, this.tileSize.width - 3, this.
                tileSize.height - 3));
18              node.setPosition(nextPosX, nextPosY);
19              node.setColor(cc.color(255, 0, 255));
20              node.setOpacity(100);
21
22              nextPosX += this.tileSize.width;
23          }
24          nextPosX = (cc.winSize.width - this.tiledMap.width) / 2 + this.tileSize.
            width / 2;
25          nextPosY += this.tileSize.height;
26      }
27 }
```

在上述代码中，第3行获取瓦片地图的地图大小，其值为cc.size(13, 7)，表示有7行13列。然后根据这个大小做了一次2层for循环，生成了对应的区域对象。

第14行到第20行为区域测试代码，其作用只是把tiledMapRectArray数组中的区域对象绘制出来。此时，再运行项目，便可以看到效果如图17-23所示。

当保存区域的数组tiledMapRectArray赋值完成之后，就需要初始化区域状态数组的默认值了，代码如下：

```
1  // 加载[瓦片地图区域映射]
2  loadTiledMapRectArrayMap : function(){
3      var i;
4      var mapSize = this.tiledMap.getMapSize();
5      for (i = 0; i < mapSize.height; i++) {
6          this.tiledMapRectArrayMap[i] = [];
7          for (var j = 0; j < mapSize.width; j++) {
8              this.tiledMapRectArrayMap[i][j] = this.tiledMapRectMapEnemu.NONE;
9          }
10     }
11 }
```

在上述代码中，第8行标记了tiledMapRectArrayMap数组中所有对象的值都为this.tiledMapRectMapEnemu.NONE，表示此位置没有任何物体。实际上，tiledMapRectMapEnemu是一个模拟枚举的对象，每个属性用来表示不同的物体，例如道路、小障碍物、大障碍物以及炮塔等，这些属性在GPMainLayer.loadProperty()函数中被初始化，代码如下：

```
1  // 加载[属性]
2  loadProperty : function(){
3      // ......
4      this.tiledMapRectMapEnemu.NONE      = 0;    // 无
5      this.tiledMapRectMapEnemu.ROAD      = 1;    // 道路
6      this.tiledMapRectMapEnemu.SMALL     = 2;    // 小障碍物[占1格]
7      this.tiledMapRectMapEnemu.LITTLE    = 3;    // 中障碍物[占2格]
8      this.tiledMapRectMapEnemu.BIG       = 4;    // 大障碍物[占4格]
9      this.tiledMapRectMapEnemu.INVALID   = 5;    // 无效区域
10     this.tiledMapRectMapEnemu.TOWER     = 6;    // 塔
11 }
```

有了区域数组（tiledMapRectArray）、区域上物体的状态标记数组（tiledMapRect-ArrayMap）以及状态表示对象（tiledMapRectMapEnemu）之后，还需要一个函数，这个函数负责接收一个点坐标，然后返回指定坐标在tiledMapRectArray中的相关信息，例如当前坐标是否在tiledMapRectArray表示的区域中、是在区域中的第几行第几列以及对应区域的坐标是多少。我将这个函数命名为getInfoFromMapByPos(x, y)，其代码如下：

```
1  // 根据坐标获取在地图格子区域中的信息
2  getInfoFromMapByPos : function(x, y){
3      cc.assert(y !== undefined,"GPMainLayer.getInfoFromMapByPos():Y坐标不能为空！");
4      var isInMap = false;  // 是否在地图中
5      var index = { x : -1, y : -1 };
6      var rect = null;
7      for (var i = 0; i < this.tiledMapRectArray.length; i++) {
8          for (var j = 0; j < this.tiledMapRectArray[i].length; j++) {
9              rect = this.tiledMapRectArray[i][j];
10             if (cc.rectContainsPoint(rect, cc.p(x, y))) {
11                 index.row = i;
12                 index.cel = j;
13                 index.x = rect.x;
14                 index.y = rect.y;
15                 isInMap = true;
16             }
17         }
18     }
19
20     return {
21         isInMap : isInMap,
22         row : index.row,  // 行
23         cel : index.cel,  // 列
24         x : index.x,
25         y : index.y
26     };
27 }
```

在上述代码中，第20行到第26行返回了指定坐标在tiledMapRectArray的信息，例如行和列。有了行和列之后，便可以在加载障碍物的同时，也在tiledMapRectArrayMap中对应的位置标记出当前障碍物的类型。例如，加载最小障碍物的代码如下：

```
1  // 加载[最小的障碍物]
2  loadSmallObstacle : function(){
3      var smallGroup = this.tiledMap.getObjectGroup("small");
4      var smalls = smallGroup.getObjects();
5      var node = null;
6      var info = null;
7      for (var i = 0; i < smalls.length; i++) {
8          node = new cc.Sprite("res/GamePlay/Object/Theme" +
           GameManager.getThemeID() + "/Object/" + smalls[i].name +".png");
9          this.addChild(node);
10         node.x = smalls[i].x + smallGroup.getPositionOffset().x;
11         node.y = smalls[i].y + smallGroup.getPositionOffset().y;
12
```

```
13          info = this.getInfoFromMapByPos(node.x, node.y);
14          this.tiledMapRectArrayMap[info.row][info.cel] = this.tiledMapRectMap
            Enemu.SMALL;
15      }
16 }
```

在上述代码中，第8行获取对象层中对象的名字，接着根据此名字，并结合当前主题在资源中找到对应的障碍物图片，然后创建出精灵节点。

第10行和第11行分别设置了障碍物精灵节点的X坐标和Y坐标。注意，它们都加上了对象层的坐标偏移量，这样保证了障碍物位置没有发生错乱。

第13行调用了`getInfoFromMapByPos(x, y)`函数，因为我们需要知道当前障碍物的位置是在`tiledMapRectArrayMap`数组中的第几行和第几列。

最后，在第14行中，对`tiledMapRectArrayMap[info.row][info.cel]`进行赋值，表示`tiledMapRectArrayMap[info.row][info.cel]`所保存的障碍物状态为最小障碍物。

除了小障碍物外，其余障碍物的加载方式都差不多，有区别的就是区域状态标记。以大障碍物为例，它占用4个格子。假设`getInfoFromMapByPos(x, y)`函数返回的行和列对应图17-24中0的位置，那么我们应该还要将0周围的3个位置都标记为大障碍物，代码如下：

```
1  // 加载[大障碍物]
2  loadBigObstacle : function(){
3      // ......
4      for (var i = 0; i < objs.length; i++) {
5          // ......
6          info = this.getInfoFromMapByPos(node.x, node.y);
7          this.tiledMapRectArrayMap[info.row][info.cel] = this.tiledMapRect
           MapEnemu.BIG;
8          this.tiledMapRectArrayMap[info.row][info.cel - 1] = this.tiledMapRect
           MapEnemu.BIG;
9          this.tiledMapRectArrayMap[info.row - 1][info.cel] = this.tiledMapRectMap
           Enemu.BIG;
10         this.tiledMapRectArrayMap[info.row - 1][info.cel - 1] = this.tiledMapRect
           MapEnemu.BIG;
11     }
12 }
```

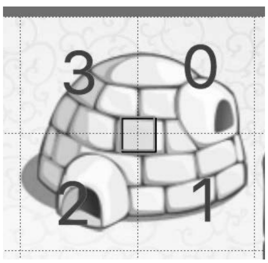


图17-24　大障碍物占用格子的情况

其他障碍物以及无效区域的标记与此类似，详见项目源码。此外，还有一个移动路径的标记，它的实现思路如下。

首先，判断当前路标点和下一个路标点的Y值是否相等，如果相等，则表示当前段移动路径为水平移动，然后再比较X坐标的大小，如果下一个路标点的X坐标大于当前路标点的X坐标，则表示水平向右运动，否则表示水平向左运动（Y轴方向上同理）。

获取出移动方向之后，再计算出当前路标和下一个路标的距离除以70的结果（格子的大小），这便是当前方向上共有的格子数量，设为offsetCount。

最后，通过一个循环次数为offsetCount的for循环，把当前方向上所有的格子都标记为tiledMapRectMapEnemu.ROAD，具体代码见GPMainLayer.loadRoadMap()函数。

此时，再运行一下项目，可以看到的效果如图17-25所示。

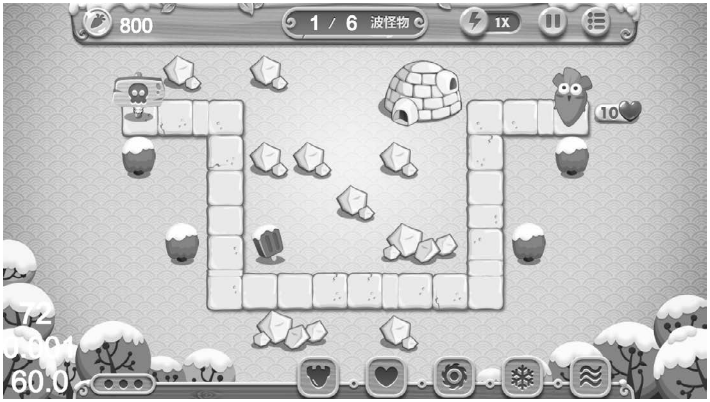


图17-25　障碍物加载

6. 创建炮塔

创建炮塔，需要给GPMainLayer注册一个TOUCH_ONE_BY_ONE触摸事件，然后在onTouchEnded(touch, event)函数中进行逻辑处理，该函数的代码如下：

```
1  onTouchEnded : function (touch, event) {
2      var self = event.getCurrentTarget();
3      var info = self.getInfoFromMapByPos(touch.getLocation().x, touch.
       getLocation().y);
4      // 没有触摸到地图区域内
5      if (!info.isInMap) {
6          self.loadTouchWarning(touch.getLocation().x, touch.getLocation().y);
7      }else {
8          // 已经有炮塔或者障碍物
9          if (self.tiledMapRectArrayMap[info.row][info.cel] != self.
           tiledMapRectMapEnemu.NONE) {
10             self.loadTouchWarning(info.x + self.tileSize.width / 2, info.y + self.
               tileSize.height / 2);
```

```
11          }else{
12              // 当前位置没有炮塔和障碍物
13              if (self.towerPanel == null) {
14                  var data = {};
15                  data.row = info.row;
16                  data.cel = info.cel;
17                  data.x = info.x;
18                  data.y = info.y;
19                  self.loadTowerPanel(data);
20              }else{
21                  self.removeChild(self.towerPanel);
22                  self.towerPanel = null;
23              }
24          }
25      }
26 }
```

在上述代码中，第3行获取当前触摸点在`tiledMapRectArray`数组所保存的区域中的情况，返回行号和列号以及是否在这片区域中，如果不在（见第5行），那么加载一个触摸警告的精灵，这个精灵运行一个组合动作，1秒之后自动删除，具体见`GPMainLayer.loadTouchWarning(x, y)`函数。

如果当前触摸的位置在`tiledMapRectArray`数组所保存的区域中，则根据返回的行号和列号去索引区域状态标记数组`tiledMapRectArrayMap`，如果索引出来的结果不是`tiledMapRectMapEnemu.NONE`（见第9行），则表示该位置有物体或者是无效触摸区域，此时同样需要创建触摸警告按钮。但是如果索引出来的结果是`tiledMapRectMapEnemu.NONE`，则表示当前位置为空，说明可以创建炮塔，此时需要构建一个`data`数据，里面保存炮塔创建成功后的X和Y坐标，以及对应的行号和列号，具体见第14行到第19行。

`GPMainLayer.loadTowerPanel()`函数的代码如下：

```
1  // 加载创建炮塔的面板
2  loadTowerPanel : function(args){
3      // 接收行号和列号
4      var node = new TowerPanel(args);
5      this.addChild(node, this.ZOrderEnum.TOWER_PANEL);
6      this.towerPanel = node;
7  }
```

在上述代码中，第4行创建出了创建炮塔要用的面板，如图17-26所示。

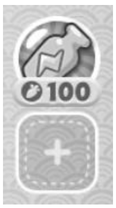


图17-26 创建炮塔面板

TowerPanel继承自精灵，然后在这个精灵之上又添加了对应炮塔的图标精灵。图标精灵需要注册一个触摸事件，在触摸事件的onTouchEnded(touch, event)函数中，会抛出一个创建塔的事件，相关代码如下：

```
1  onTouchEnded: function (touch, event) {
2      // target指向对应炮塔的图标
3      var target = event.getCurrentTarget();
4      // 创建炮塔事件分发
5      var createTowerEvent = new cc.EventCustom(jf.EventName.GP_CREATE_TOWER);
6      createTowerEvent.setUserData({
7          name : target.getName(),
8          // target.getParent()指向TowerPanel
9          x : target.getParent().getPositionX(),
10         y : target.getParent().getPositionY(),
11         cel : target.getParent().cel,
12         row : target.getParent().row
13     });
14     cc.eventManager.dispatchEvent(createTowerEvent);
15 }
```

可以看到，第5行创建了一个名为创建炮塔的自定义事件，第6行到第13行配置了事件携带的数据对象，第14行将事件抛出。

毫无疑问，创建炮塔的事件应该在GPMainLayer.registerEvent()函数中被注册监听，代码如下：

```
1  // [事件监听]创建炮塔的事件
2  var onCreateTowerListener = cc.EventListener.create({
3      event       : cc.EventListener.CUSTOM,
4      target      : this,
5      eventName   : jf.EventName.GP_CREATE_TOWER,
6      callback    : this.onCreateTower
7  });
8  cc.eventManager.addListener(onCreateTowerListener, this);
```

当GPMainLayer收到创建炮塔的事件之后，会调用onCreateTower(event)函数，该函数获取出事件对象event所携带的数据，并通过此数据创建出对应的炮塔，代码如下：

```
1  // [事件]创建炮塔
2  onCreateTower : function(event){
3      var self = event.getCurrentTarget();
4      var data = event.getUserData();
5      // 根据数据创建出炮塔
6      var node = self.createTower(data);
7      self.addChild(node);
8      // 移除创建炮塔的面板
9      self.removeChild(self.towerPanel);
10     self.towerPanel = null;
11 }
```

在上述代码中，第6行根据data数据创建泡塔，第7行将炮塔添加到GPMainLayer中。createTower(data)函数的代码如下：

```
1  // 创建炮塔
2  createTower : function(data){
3      cc.assert(data.name, "GPMainLayer.createTower(): 名字无效! ");
4      cc.assert(data.x, "GPMainLayer.createTower(): X轴坐标无效! ");
5      cc.assert(data.y, "GPMainLayer.createTower(): Y轴坐标无效! ");
6
7      var towerData = {};
8      towerData.name = data.name;
9      towerData.x = data.x;
10     towerData.y = data.y;
11
12     var node = null;
13     switch (data.name){
14         case "Bottle":
15             towerData.scope = 300;
16             towerData.bulletSpeed = 20;
17             node = new Bottle(towerData);
18             break;
19         default :
20             cc.warn("GPMainLayer.createTower() : 异常");
21             break;
22     }
23
24     if (node != null) {
25         // 标记当前位置有炮塔
26         this.tiledMapRectArrayMap[data.row][data.cel] =
           this.tiledMapRectMapEnemu.TOWER;
27     }
28     return node;
29 }
```

在上述代码中，第7行到第17行进一步配置了炮塔所需要的数据对象，从而创建出了炮塔。然后第26行将当前炮塔所在的区域位置标记为`tiledMapRectMapEnemu.TOWER`。

第17行创建出了瓶子炮塔对象，下面就来看看炮塔类应该如何设计。

7. 炮塔的设计

从《保卫萝卜2》游戏中可以知道，每一个瓶子炮塔都会朝最近的敌人发射子弹。经过分析可以得出，瓶子炮塔至少需要炮塔的视野、子弹的速度以及子弹最终要发射的位置这3个属性，除此之外，子弹最终的位置又取决于最近的敌人，所以还需要一个属性来保存对中的敌人。再者，炮塔有个底座，底座上面有个可以旋转方向的炮，所以还需要定义一个属性来指向这个炮。但是，为了让炮塔类具有良好的扩展性，我编写了一个炮塔的父类`TowerBase`（位于src/Scene/GamePlay/Node/Tower/TowerBase.js文件中），这些属性都放在父类中，代码如下：

说明 在《保卫萝卜2》中有多种炮塔，这里我不打算完成所有炮塔的开发，仅选择瓶子炮塔作为示例讲解。

```
1  var TowerBase = cc.Sprite.extend({
2      scope           : 0,            // 视野
3      bulletSpeed     : 10,           // 子弹速度
4      bulletMoveTime  : 0,            // 子弹移动到终点所需的时间
5      nearestEnemy    : null,         // 最近的敌人
6      fireTargetPos   : cc.p(0, 0),   // 子弹最终的位置
7      weapon  : null,      // 武器
8      row     : -1,        // 行
9      cel     : -1,        // 列
10     ctor : function(fileName, data){
11         this._super(fileName, data);
12     }
13 });
```

构造函数ctor（见第10行）中的参数fileName为炮塔底座图片，而data则为炮塔的配置数据。显然，我们需要一个函数来读取出data中所有的数据，其代码如下：

```
1  // 加载[属性]
2  loadProperty : function(data){
3      cc.assert(data.scope > 0, "TowerBase.loadProperty() : scope必须大于0！");
4      cc.assert(data.bulletSpeed > 0, "TowerBase.loadProperty() : bulletSpeed
       必须大于0！");
5      this.scope = data.scope;
6      this.bulletSpeed = data.bulletSpeed;
7      this.bulletMoveTime = 100 / this.bulletSpeed;
8      this.setPosition(data.x, data.y);
9      this.setName(data.name);
10 }
```

除此之外，我还给TowerBase添加了一个findNearestEnemy()函数，用来从GameManager.currMonsterPool数组中找出当前炮塔最近的对象，并返回此怪物对象，其代码如下：

```
1  // 找到最近的敌人
2  findNearestMonster : function(){
3      var monsterArray = GameManager.currMonsterPool;
4      var currMinDistant = this.scope;
5      var nearestEnemy = null;
6      var monster = null;
7      var distance = 0;
8      for (var i = 0; i < monsterArray.length; i++) {
9          for (var j = 0; j < monsterArray[i].length; j++){
10             monster = monsterArray[i][j];
11             distance = cc.pDistance(this.getPosition(), monster.getPosition());
12             if (distance < currMinDistant) {
13                 currMinDistant = distance;
14                 nearestEnemy = monster;
15             }
16         }
17     }
18     this.nearestEnemy = nearestEnemy;
19     return nearestEnemy;
20 }
```

然后借助面向对象编程的多态特性，给TowerBase添加loadWeapon()和onFire()函数，

分别用来加载武器和开火，这两个函数需要在子类中重写，其代码如下：

```
1  // 加载[武器]
2  loadWeapon : function(){
3      cc.warn("TowerBase.loadWeapon() : 请重写此方法！");
4  },
5  // 开火
6  onFire : function(){
7      cc.warn("TowerBase.onFire() : 请重写此方法！");
8  }
```

炮塔父类TowerBase开发完成后，便可以进一步开发出瓶子炮塔Bottle了。显然，Bottle直接继承自TowerBase。此外，瓶子炮塔每隔一段时间就会发射一颗子弹，所以，在Bottle的构造函数里，我开启了一个调度器，代码如下：

```
1  var Bottle = TowerBase.extend({
2      ctor : function(data){
3          this._super("#Bottle_3.png", data);
4          // 0.5秒开火一次
5          this.schedule(this.onRotateAndFire, 0.5);
6      },
7  });
```

在上述代码中，第5行开启了一个调度器，每隔0.5秒就会调用一次onRotateAndFire()函数。在onRotateAndFire()函数中，首先会调用this.findNearestMonster()函数找到最近的怪物，然后计算出当前炮塔和怪物的距离以及角度等，最终对怪物开火，其代码如下：

```
1  // 旋转并开火
2  onRotateAndFire : function(){
3      var nearestEnemy =  this.findNearestMonster();
4      if (nearestEnemy != null){
5          this.weapon.stopAllActions();
6
7          this.fireTargetPos = nearestEnemy.getPosition();
8
9          var rotateVector = cc.pSub(nearestEnemy.getPosition(), this.
           getPosition());
10         var rotateRadians = cc.pToAngle(rotateVector);
11         // 弧度转为角度
12         var rotateDegrees = cc.radiansToDegrees(-1 * rotateRadians);
13
14         // speed表示炮塔旋转的速度，0.5 / M_PI其实就是 1 / 2PI，它表示1秒钟旋转1个圆
15         var speed = 0.5 / cc.PI;
16         // rotateDuration表示旋转特定的角度需要的时间，计算它用弧度乘以速度
17         var rotateDuration = Math.abs(rotateRadians * speed);
18
19         var move = cc.rotateTo(rotateDuration, rotateDegrees);
20         var callBack = cc.callFunc(this.onFire, this);
21         var action = cc.sequence(move, callBack);
22         this.weapon.runAction(action);
23     }
24 }
```

在上述代码中，第7行保存了下一颗子弹要发射到的最终位置。

第9行用最近怪物的坐标减去当前炮塔的坐标，从而得到一个向量坐标。然后在第10行中，将rotateVector向量坐标转为弧度，这个弧度再通过cc.radiansToDegrees(radians)函数转为角度，并且弧度需要乘以-1，这是因为Cocos中规定顺时针方向为正，而在数学中，顺时针为负，所以转化的时候需要把角度a变为-a。

第15行定义了炮架转动的速度，第17行计算出了炮架转向到最近的怪物所需要的时间，第19行到第22行构建并运行了一个组合动作，从而实现向炮架开火。

需要注意的是，第20行中的onFire函数。在该函数中，会创建出子弹对象，每颗子弹会运行一个移动到终点，然后调用removeBullet()函数的组合动作（代码详见随书源码）。

removeBullet()函数用来移除当前子弹对象，在该函数内部，它会抛出一个移除子弹的事件，并把子弹自身作为事件携带对象一并抛出。显然，这个事件也是在GPMainLayer.registerEvent()函数中注册监听。removeBullet()函数的实现代码如下：

```
1  removeBullet : function(sender){
2      var event = new cc.EventCustom(jf.EventName.GP_REMOVE_BULLET);
3      event.setUserData({
4          target : sender
5      });
6      cc.eventManager.dispatchEvent(event);
7  }
```

此时再运行项目，然后在地图中创建几个瓶子炮塔，可以看到效果如图17-27所示。

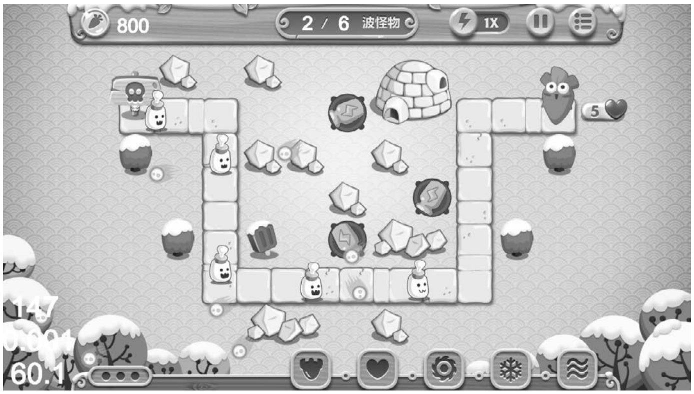


图17-27　创建炮塔

虽然炮塔已经可以转动并发射子弹，但是子弹打中怪物之后，怪物并不会死，这是因为我还没有做子弹和怪物之间的碰撞检测。

8. 子弹和怪物的碰撞检测

子弹和怪物的碰撞检测，需要在GPMainLayer中开启update调度器，然后在GPMainLayer.

update()函数中调用子弹和怪物的碰撞检测函数checkCollide()。checkCollide()函数的逻辑代码如下：

```
1  // 子弹和怪物的碰撞检测
2  checkCollide : function(){
3      var bullet = null;
4      var monster = null;
5      var monsterRect = null;
6      for (var x = 0; x < GameManager.currBulletPool.length; x++) {
7          for (var y = 0; y < GameManager.currMonsterPool.length; y++) {
8              for (var z = 0; z < GameManager.currMonsterPool[y].length; z++) {
9                  // 获取子弹
10                 bullet = GameManager.currBulletPool[x];
11                 // 如果在JSB上被回收，则跳过
12                 if (!cc.sys.isObjectValid(bullet)) {
13                     break;
14                 }
15                 monster = GameManager.currMonsterPool[y][z];
16                 monsterRect = cc.rect(monster.x - monster.width / 2, monster.y -
                   monster.height / 2, monster.width, monster.height);
17                 if (cc.rectContainsPoint(monsterRect, bullet.getPosition())) {
18                     // 移除子弹
19                     this.removeBulletByIndex(x);
20                     // 移除怪物
21                     this.removeMonsterByIndex(y, z);
22                     // 是否进入下一组
23                     if (this.isNeedLoadNextGroup()){
24                         this.loadNextGroupMonster();
25                     }else{
26                         if (GameManager.getGroup() > GameManager.getMaxGroup()) {
27                             var event = new cc.EventCustom(jf.EventName.GP_GAME_
                               OVER);
28                             event.setUserData({
29                                 isWin : true
30                             });
31                             cc.eventManager.dispatchEvent(event);
32                         }
33                     }
34                 }
35             }
36         }
37     }
38 }
```

可以看到，第6行到第8行执行了一个3层for循环，其实现思路是取出一颗子弹，然后再去遍历GameManager.currMonsterPool二维数组，因为这个二维数组中存放了所有的怪物。但是，需要注意的是子弹的获取，并不是在第一层for循环中，而是在第三层for循环里（具体见第10行）。这是因为如果子弹和怪物碰撞了，则在第19行中通过调用removeBulletByIndex(index)函数删除子弹，此时会改变GameManager.currBulletPool数组的长度。

另外需要注意的是第12行到第14行。在JSB中，如果当前子弹已经被移除，那么直接跳过当

前循环。cc.sys.isObjectValid(obj)是一个非常好用的函数，它用来在JSB中检测节点的有效性，从而避免Invalid Native Object错误。

第17行判断了子弹是否碰撞到怪物，如果碰撞到了，那么移除子弹（见第19行）、移除怪物（见第21行），并且判断是否需要进入下一组，如果需要，则加载下一组怪物，如果不需要，再判断是否是最后一组怪物，如果是，则表示游戏已经结束。游戏结束的情况有两种——成功和失败，这里我也采用抛事件的方式，抛出一个游戏结束事件，并在事件中携带着游戏输赢的状态。

游戏结束事件的回调函数为GPMainLayer.onGameOver()，其代码如下：

```
1  // [事件]游戏结束
2  onGameOver : function(event){
3      var data = event.getUserData();
4      GameManager.setIsWin(data.isWin);
5      cc.audioEngine.stopMusic();
6
7      var scene = new GameResultScene();
8      cc.director.runScene(scene);
9      var str = data.isWin ? "赢了！" : "输了";
10     cc.log("GPMainLayer.onGameOver() : 游戏结束，你" + str);
11 }
```

这里需要注意第4行，它将游戏胜利或者失败的状态保存在GameManager.isWin属性中，然后在第7行中创建了游戏结束场景，并在第8行做了场景跳转。需要说明的是，游戏结束场景中的代码，也只是一些UI的组合，这里我不再展开讲解，你可以自行查阅随书代码。

最后，把之前留在关卡选择场景中的TODO任务完成，使得整个游戏Demo完整地连接在一起。整个TODO任务在关卡选择页面背景层的CLBackgroundLayer.onLevelButtonEvent()函数中，其代码如下：

```
1  // 事件[关卡按钮]
2  onLevelButtonEvent : function(sender, type){
3      switch (type) {
4          case ccui.Widget.TOUCH_ENDED:
5              var level = sender.getTag();      // 当前等级
6              // TODO：加载关卡数据，进入游戏
7              // 停止音乐
8              cc.audioEngine.stopMusic();
9              // 关卡设置
10             GameManager.setLevel(sender.getTag());
11             // 加载资源，并进入游戏
12             cc.LoaderScene.preload(g_gamePlay_resources, function () {
13                 GameManager.loadLevelData(GameManager.getLevel());
14                 cc.director.runScene(new GamePlayScene());
15             }, this);
16         break;
17     }
18 }
```

至此，《保卫萝卜2》项目实战的Demo开发完成，此时再运行一下项目，进入到游戏玩法

场景，建立几个炮塔，成功击退了所有的怪物之后，便可以看到游戏结束的胜利场景如图17-28所示。



图17-28　游戏胜利

最后需要说明的是，这里我仅仅演示了第一关的开发，但是实际上，在随书源码中，我配置了3个关卡的数据，以及编辑出了3个关卡的地图配置。

另外，在《保卫萝卜2》的实际项目中，点击某一个障碍物之后，最近的炮塔会攻打此障碍物，最后把障碍物移除掉，这一功能留作本书唯一的作业，如果你能在基于本书随书代码完成这一功能，那么你便完全理解Carrot项目的代码设计了。

17.4　小结

通过本章的学习，我们完成了《保卫萝卜2》项目Demo的开发。在这个项目中，我们学习了关卡数值数据应该写在一个单独的配置文件中，而其他配置（例如障碍物等关卡配置）可以通过Tiled Map Editor编辑器中的对象层标记。每一个关卡的数值数据可以放在GameManager对象中管理，然后GameManager只提供一些简单的接口，保证游戏结构清晰，数据不会混乱。另外，我们还学习一套两个数组配合使用的映射标记法，即一个二维数组中保存区域位置，另一个二维数组中保存另一个数组中对象的状态信息，这在游戏开发中也是一种常用的方法。例如，在游戏中实现擦黑板，擦除85%面积以上，则表示擦除完毕，便可以通过此组合方式来实现。



从建风的教学视频就开始关注他，感谢他和Cocos一起成长。这本呕心沥血之作涵盖面广，由浅入深，甚至用商业大作《保卫萝卜2》的资源来寓教于乐，从各个方面都可以说是非常难得的。

——panda（凌华彬），Cocos2d-JS引擎核心开发者

Cocos是业内领先的一站式跨平台手游开发解决方案，市面上已经有大量的教程和图书来介绍如何使用Cocos，但是这些书和教程大多数使用的是C++和Lua，鲜有比较完整且深入介绍Cocos2d-JS的图书，本书弥补了这方面的空白。建风耗费无数个午夜埋头撰写书稿，在初稿完成之际也邀请了我进行技术审稿，我可以毫不夸张地说，这绝对是一本良心之作，推荐给大家。

——子龙山人（屈光辉），Cocos2d-x引擎核心开发者

目前市面上使用C++、Lua语言进行Cocos游戏开发的教程颇多，但缺少基于JavaScript来开发的系统教程。建风作为知名Cocos技术专家，在本书的编写上花费了颇多精力，今得知本书横空出世，相信将填补这部分技术教程的空白。随着手机游戏页游化进程的推进和Cocos Creator工具的推出，基于JavaScript的开发模式将会更加受欢迎。本书内容丰富而实用，可作为广大对这些知识感兴趣的Cocos爱好者学习和参考之用。

——红孩儿（卞安）

学习要学以致用，教学应授人以渔。建风将其恰到好处地融入本书中，在详细讲解各知识点的同时又结合项目实例的引导方式，让读者更透彻地理解与吸收！这着实是一本良心之作，值得大家入手。

——Himi（李华明）

建风是触控科技认证CVP，是CVP平台年轻有为的Cocos2d-JS体系专家，对Cocos2d-JS引擎理解深刻，开发经验丰富，且先后在CocoaChina社区和CVP平台上发布两套Cocos2d-JS视频教程，均获得超高好评。本书内容全面、重点突出，不仅是初级开发者的入门教程，也可以帮助更多游戏开发者全面掌握引擎和游戏开发技巧。社区和平台对这本书也非常期待，并会鼎力支持和推广！

——CocoaChina社区和CVP平台

手机游戏发展至今，产品层出不穷，而这其中不乏优秀的游戏产品脱颖而出，每一款产品都是游戏开发人员辛勤汗水的浇筑，技术是构建游戏的基本环节，本书正是建风在实际开发工作中的经验累积。如今建风将这些经验毫无保留地分享给大家，可以帮助大家了解更多的游戏开发经验和技巧，如果你对Cocos2d-JS游戏开发有兴趣，那么请不要错过这本好书！

——林德辉，GameRes游资网CEO

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/游戏开发

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-42148-7

ISBN 978-7-115-42148-7
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks